

Technical Report
CMU/SEI-94-TR-8
ESC-TR-94-008
May 1994

ADA283747

Mapping a Domain Model and Architecture to a Generic Design



**A. Spencer Peterson
Jay L. Stanley Jr.**

Application of Software Models Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

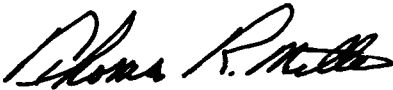
SEI Joint Program Office
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1994 by Carnegie Mellon University

This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994.

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1	Introduction and Background	1
1.1	Audience	2
1.2	Purpose	2
1.3	Overview of the Movement Control Domain	3
1.4	Report Overview — How to Read This Document	4
2	Context for the Mapping Process	7
2.1	Using Models in Software Development	8
2.2	The Domain Model — FODA Products and Representations	10
2.3	The Architecture — The OCA	13
2.3.1	Overview of the OCA	14
2.3.2	OCA Components	14
2.3.2.1	Objects	14
2.3.2.2	Controllers	15
2.3.2.3	Imports	16
2.3.2.4	Exports	16
2.3.2.5	Signatures	16
2.3.2.6	Surrogates	18
2.3.2.7	Executives	19
2.3.3	Flow of Control and Data in the OCA	19
2.4	The Generic Design	21
3	Overview of the Mapping Process	23
3.1	Partitioning the Process	23
3.2	Viewing the Process as a Normal S/W Development Process	25
3.3	Use of the OCA in the Development of Reusable Software	26
3.4	Benefits from Reusing OCA Structures	26
3.4.1	Consistency of Form Within Applications	26
3.4.2	Separation of Control Flow and Data Flow	27
3.5	Limitations of the Mapping Process	27
3.6	A Roadmap for the Details of the Mapping Process	28
4	The Domain Design Process	31
4.1	Select Features from Domain Model	33
4.2	Create Object Specifications	34

4.2.1	Identify Objects	34
4.2.2	Derive Object Operations and Input/Outputs	35
4.3	Create the Subsystem Specifications	36
4.4	Create a Surrogate Specification for Each Logical/Physical Device	37
5	The Domain Implementation Process	39
5.1	Identify or Create Applicable System Engineering Units Package(s)	41
5.2	Create Subsystem <i>_Types</i> Package	42
5.3	Create Object Signatures Package	43
5.4	Create Object Manager Package Specification	43
5.5	Create Subsystem/Surrogate Signatures Package	44
5.6	Create Subsystem/Surrogate Controller Package Specification	45
5.7	Create Subsystem/Surrogate Import and Export Packages	46
5.8	Create Subsystem/Surrogate Controller Package Body	47
5.9	Create Object Manager Package Body	48
6	Application Development Using a Generic Design	51
6.1	Create an Application Signatures Package	51
6.2	Complete Packages Making Use of Application Signatures	52
6.2.1	Complete Surrogate Signatures Package	52
6.2.2	Complete Surrogate Controller Package Specification	52
6.2.3	Complete Surrogate Import/Export Packages	53
6.2.4	Complete Subsystem Import Package Body	53
6.2.5	Complete Surrogate Controller Package Body	53
6.3	Complete the Executive Template	53
7	Conclusions and Future Directions	55
7.1	Conclusions	55
7.2	Future Directions	56
7.2.1	Near-Term	56
7.2.2	Long-Term	56
	Appendix A The Domain Design Process	63
	Appendix B The Domain Implementation Process	65
	Appendix C Using a Generic Design in Application Development	71

Appendix D Specification Form Templates	75
Appendix E Ada Code Templates	79
Appendix F Implementation Issues Affecting Reuse	91
Appendix G Sample Completed Specification Forms	97
Appendix H Movement Control Example Code	103

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

List of Figures

Figure 1-1:	Roadmap for the Mapping Process	3
Figure 2-1:	The Mapping Process — Its Inputs and Outputs	7
Figure 2-2:	Use of Models in an Engineering Framework	9
Figure 2-3:	Example of an OO1 Information Model	11
Figure 2-4:	Example Representation of an OO1 Features Model	12
Figure 2-5:	The OCA Subsystem Model	15
Figure 2-6:	Signatures as Conceptual Links to Object Functionality	18
Figure 2-7:	Overall Flow of Data and Control in the OCA	20
Figure 2-8:	Convoy Planner Generic Design	21
Figure 3-1:	Decomposition of the Mapping Process	23
Figure 3-2:	Migration of Domain Data to Specifications and Code	24
Figure 3-3:	Mapping Design Elements to Process Steps	29
Figure 5-1:	Subsystem Implementation Model	49
Figure 7-1:	A Development Life Cycle Utilizing the Mapping Process	55
Figure F-1:	Tasking Architecture Using a Separate X Event Loop	93

List of Tables

Table 4-1:	Summary of the Domain Design Process	31
Table 4-2:	Mapping Domain Model Constructs to Specification Forms	32
Table 5-1:	Summary of the Design Implementation Process	39
Table 5-2:	Mapping from Specification Forms to Code Constructs	41
Table 6-1:	Summary of the Application Development Process	51

Mapping a Domain Model and Architecture to a Generic Design

Abstract: In contrast to the number of reports on domain analysis, little work has been done in describing the utilization of domain analysis results in the development of generic designs for building applications in a domain. This report describes a process for mapping domain information in Feature-Oriented Domain Analysis (FODA) into a generic design for a domain. The design includes supporting code components that conform to the Object Connection Architecture (OCA), a model for structuring software systems. A process for the use of the design in implementing applications is included. The processes and products described herein augment the final phase of domain analysis (or engineering) described in the original FODA report. This report also documents the continuing work of applying FODA to the movement control domain. The design and Ada code examples for the domain used in the document are from prototype software, created in part to test the processes presented.

1 Introduction and Background

There has been a significant amount of research in the area of domain analysis. [Prieto-Diaz 91] provides an excellent introduction into the state of domain analysis as a software engineering activity. One important aspect that has been virtually untouched in the pertinent literature is: how does one select and/or develop a design for use in building applications from the products of domain analysis? Nearly all domain analysis methods either do not address this issue at all or assume there is a design to be (re)used from the existing system(s) analyzed. There is no notion of a generic design that reflects the allocation of capabilities to subsystems or components at the logical level that is:

- independent of implementation considerations such as centralized versus distributed processing, and
- usable for all systems to be built and maintained within a program family¹ from the domain.

This report describes our efforts in this area, which are founded upon the following two premises:

1. A domain model, the product of domain analysis, embodies the requirements for software in a domain.²

¹ The term "program family" is used as defined in [Parnas 76].

² The Feature-Oriented Domain Analysis (FODA) method, developed by the Software Engineering Institute (SEI), is one domain analysis method. It captures and organizes information (especially the requirements) from existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology. FODA emphasizes the understanding of the commonalities and differences in previous and anticipated systems in that domain. The pertinent processes and products of FODA are described briefly in Section 2.2. A more complete description of FODA is given in [Kang 90].

2. Software architectures³ exist that provide a framework for generic designs. Generic designs increase the reusability of software components implemented to fit within that design by creating patterns for the components.

This report describes a process for mapping domain information captured in FODA models into a generic design for software in a domain. The Object Connection Architecture (OCA) is the architectural model used to structure the generic design. The use of the OCA in structuring software systems is described briefly in Section 2.3 of this report and will be fully documented in a subsequent report.

1.1 Audience

This report is intended to support current and future users of the FODA method of domain analysis in their efforts to produce reusable software assets at the design and large scale component level. Software architects and designers (such as the Core Asset team referred to in [Withey 94]) will derive the most benefit from the report, as they will be following the process and creating the assets. Domain analysts will need to be cognizant of the process described in Chapters 4 and 5 to understand the utilization of the information gathered in the domain analysis process.

This report also documents the SEI's efforts to utilize the processes and procedures described herein for the development of prototype software for the Army movement control domain from FODA models documented in [Cohen 92].

This report is one of four reports which further document the FODA method and its use. These reports are products of the SEI's continued work in domain analysis and its application within the software development lifecycle. The other three reports are:

1. *Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology* [Krut 93],
2. *A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis* [Fernandez 93], and
3. *Implementing Model-Based Software Engineering in Your Organization: An Approach to Domain Engineering* [Withey 94].

1.2 Purpose

This report delineates a process and products which satisfy the intent of the FODA Architectural Modeling process and migrates the use of FODA products into the design and implementation of code. This migration is illustrated in Figure 1-1. It shows that the mapping process uses domain model and architecture information to produce a generic design that, in turn, is used in an application development process to produce application code.

³ The term *software architecture* is defined in Section 2.3.

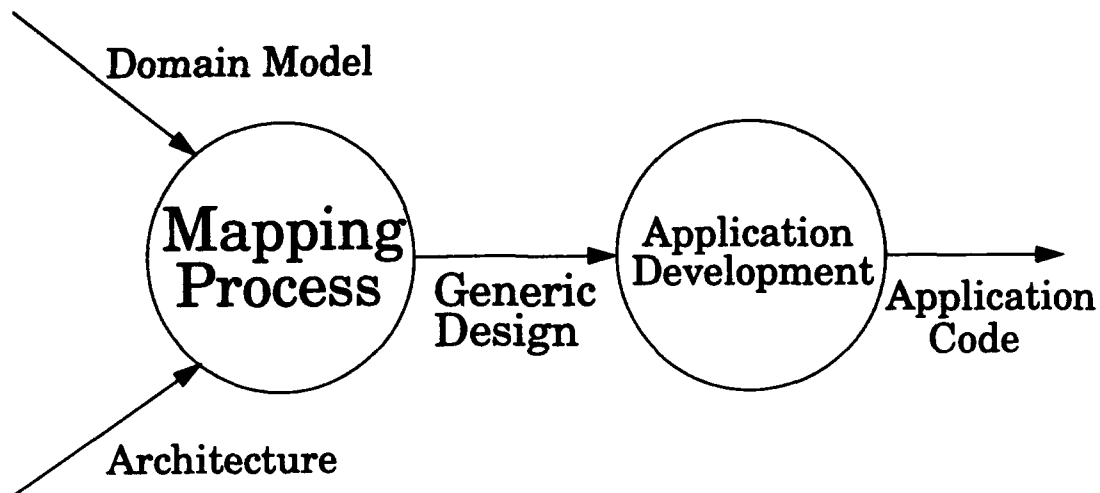


Figure 1-1: Roadmap for the Mapping Process

Although FODA products are assumed to be the inputs to the processes described in this reports, usable results may be possible through use of the products of other domain analysis methods. The alternative method used must capture the equivalent information contained in FODA Domain Model products such that persons attempting to follow the processes can locate and use the specific inputs for each step. The resulting software structures may be implemented in many popular programming languages, such as Ada, C, C++, and PASCAL. The Ada programming language is used in the software examples described in this document.

This report:

- demonstrates the concept of generic designs for program families,
- provides practical guidance for the development of such designs, and their use in building software systems, and
- advances the state of the practice in Domain Engineering and software architectures.

The mapping process described herein is intended for use by software engineers who need to develop a reusable software design and code implementation using FODA product models as the basis for requirements to be satisfied by software systems in a domain.

1.3 Overview of the Movement Control Domain

Before going into the mapping process in any detail, it is appropriate to provide a brief overview of the domain from which the examples in the subsequent chapters and appendices are derived.

Movement Control is *the planning, routing, scheduling, control, and in-transit visibility of personnel, units, equipment, and supplies moving over lines of communication in accordance with the directives of command planning* [USArmy 90]. The most common application within the this domain used by the majority of Army units is **Convoy Planning**. The operational features needed to provide convoy planning capabilities include:

- **Convoy Building** - selecting the vehicles for use in transporting whatever is to be moved and organizing them into a convoy.
- **Routing** - selecting a route using the available road network (and potentially off-road paths), taking into account the capabilities and characteristics of the vehicles involved.
- **Scheduling** - determining the travel time for a given convoy and route combination, accounting for additional stops as required.

Important data entities for these features within convoy planning include:

- **Units** - encompassing personnel, equipment, etc.
- **Road Network** - a structure containing information about points of interest and the roads between them.
- **Schedules** - a structure containing information about events, where an event is a combination of a time and an occurrence of interest.

[Cohen 92] provides a comprehensive description of the movement control domain model. This description has been given to enable the reader's understanding of more specific issues in the movement control domain used as examples to illustrate important concepts in the mapping process.

1.4 Report Overview — How to Read This Document

The remainder of this report is organized as follows:

Chapter 2 lays the foundation for the mapping process by:

1. describing the mapping process in terms of
 - a. the application of various classes of software models, and
 - b. how other software engineering processes can apply the different classes of models in obtaining their results.
2. giving a brief description of the Domain Modeling phase of FODA, concentrating on the products of interest derived during that phase and the representation of those products.
3. describing the OCA in terms of its structures and concepts.

Chapter 3 presents the mapping process in terms of the domain model information used, the products generated, and the applicability of the products to the development process.

Chapter 4 presents the Domain Design process for developing specifications for reusable domain-specific abstractions from information captured in FODA models.

Chapter 5 presents the Domain Implementation process for mapping those specifications onto the OCA as a generic design with supporting code components.

Chapter 6 describes the Application Development process for the creation of an application using components built as described and an executive built using a standardized template.

Chapter 7 presents a brief set of conclusions and a discussion of future directions for the mapping process.

In addition to the material in the main body of the report, there are 8 appendices whose contents are described below:

1. Appendix A presents the details of the Domain Design process via the completion of prescribed forms.
2. Appendix B presents the details of the Domain Implementation process via creating code units that satisfy the previous specifications through the mapping of form information onto various code constructs.
3. Appendix C presents the details of the Application Development process for the use of the generic design and its components in the creation of an instance that satisfies specific requirements.
4. Appendix D lists the Specification Forms for the Subsystem, Object and Surrogate abstractions described in the OCA.
5. Appendix E lists the code templates for implementing the OCA abstractions using the Ada programming language.
6. Appendix F discusses some of the implementation issues dealt with during the trial usage of these processes, focusing on Ada language interface issues, and the idiosyncrasies found in implementations of Ada input/output packages. It also provides some specific examples of "C" code used in the user interface portion of the movement control prototype used as the example case in the report, focused mainly on the description of several reusable abstractions for X/Motif input and output.
7. Appendix G provides examples of completed specification forms for an example subsystem, object, and surrogate from the Army movement control domain.
8. Appendix H provides an extensive sample of code from the Army movement control domain as empirical evidence of the viability of the processes presented in this report.

2 Context for the Mapping Process

The mapping process for moving from domain models to generic designs is illustrated in Figure 2-1, in SADT⁴ form. The major input is the domain model, with its collective information about the capabilities, data organization, and processing flow for systems in the domain. The architecture is a *control* input, because it structures the output, the generic design. The major resources required are the time of the domain engineers to perform the process and the tools they use to capture the results.

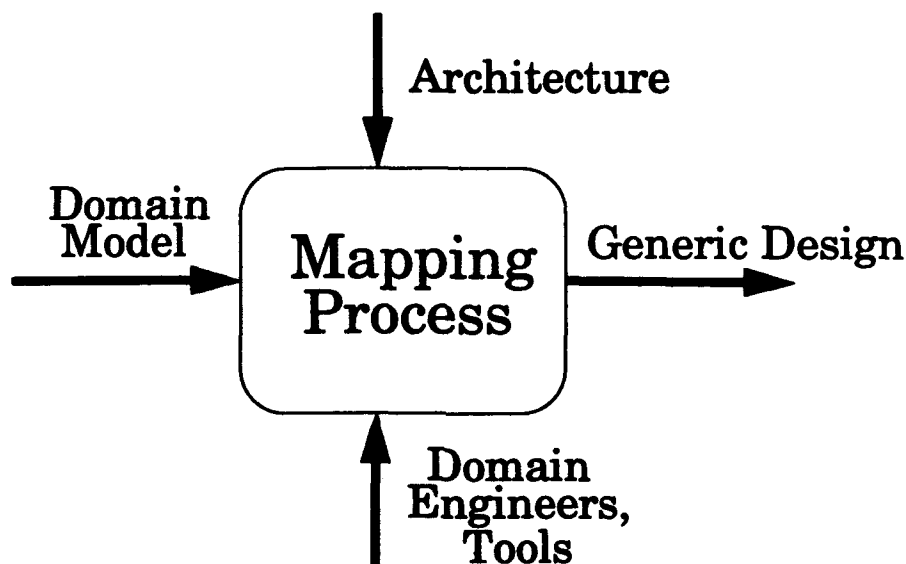


Figure 2-1: The Mapping Process — Its Inputs and Outputs

The mapping process is a series of both synthesis and analysis steps, which is broken into two major groupings:

1. Analysis of the domain model and its contents to find:
 - a. the major physical or logical abstractions that maintain state, the domain *objects*, and
 - b. the group of related features that describe the *subsystems* which utilize the objects in their implementation.

The subsystems and objects are specified using forms (described in the report) to collect the applicable information from the domain model structures.

2. Mapping of the subsystem and object specifications onto code templates, using the information collected or referenced on the specification forms to complete the templates.

⁴ See [Marca 88] for a complete explanation of the SADT notation.

This chapter first provides the reader with a brief explanation of the basic theory for software engineering with models and its applicability to the mapping process. Then the reader is introduced to the models integral to the mapping process, the FODA domain model and its products and the Object Connection Architecture.

2.1 Using Models in Software Development

Application of the FODA method results in various products, most of which are expressed in the form of models. Just as models are the basis for describing domain information, models should be the basis for describing software designs and for the performance of software engineering tasks in general. A **software model**⁵ is:

1. *A view of a domain consisting of abstractions important for analyzing and implementing a capability planned for a software system.*
2. *A representation of a system that focuses on a single concern, usually by simplifying detail.*

There are various kinds of models that can be defined for the engineering of software:

- **Abstract model** - *A set of concepts, principles, and rules used to prescribe the structure, allowable content, and key properties of a concrete model. The set is constructed with the expectation that, through use of the model, structure and behavior can be added to create concrete models.*

The notion of an abstract model is equivalent to that of an abstract class in Object Oriented Design, such as described in [Booch 93]. An abstract model is incomplete when initially defined. It requires the insertion of domain information to be fully defined.

Abstract models include meta-level concepts that are independent of any domain. Examples include:

- the notions of Aggregation/Decomposition, Generalization/Specialization and Parameterization, used as the guiding principles for the processes and products of FODA.⁶
- the use of consistent form and the various '-ilities' (understandability, modifiability, etc.) of software designs and code.
- **Concrete model** - *A view of a domain that organizes domain information in elements that encapsulate differences among existing and/or potential implementations (members of the program family).*
- **Product** - *A software system delivered to a customer which contains instances of concrete models.*

⁵ The definitions on this section are taken from [Willey 94].

⁶ See Section 3.1 of [Kang 90] for a more thorough explanation to the concepts used.

This definition of product is not meant to preclude the DoD view of software deliverables, which includes the development and delivery of specification and design documentation as interim products. A software specification can be delivered containing an instance of a domain model and a design document can contain an instance of a generic design.

A SADT diagram showing the use of models in a software process is given in Figure 2-2. It shows that a model-based software process is the result of:

- using previous information or a model as input,
- applying a model at a higher conceptual level than the input as a control, and
- producing as output a model at the same conceptual level as the input.

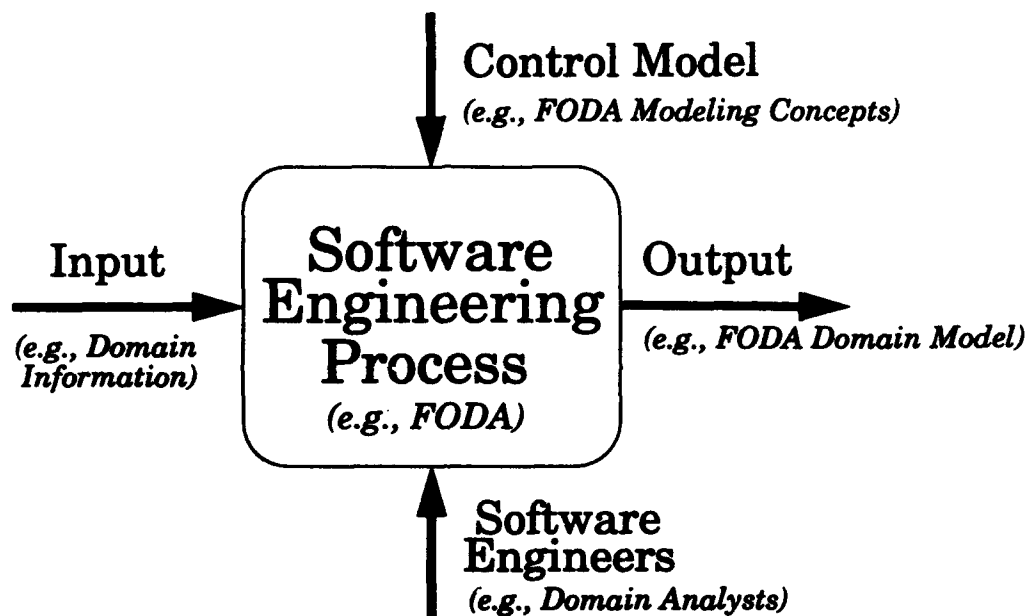


Figure 2-2: Use of Models in an Engineering Framework

This generalized model-based process is the conceptual basis for an overall software engineering life cycle entitled *Model-Based Software Engineering (MBSE)*, a concept first described by the SEI in [Feller 93]. MBSE enables organizations to build software applications which must evolve with a minimum of rework and scrap to meet changes in mission and technology. MBSE involves building models of the requirements and design for a family of software applications. Application generators and component libraries that support the software models are also built. MBSE is a focus area for the SEI's Engineering Techniques Program and is the subject of a recent SEI report [Withey 94].

The usage and reification of models from abstractions to domain specific concrete models and on into delivered software products is the fundamental process in MBSE and occurs in many forms. Concrete models are created from the application of abstract models, and products are derived from concrete models. As an example, FODA is the application of domain modeling concepts (at the abstract level) to information on existing systems and new technologies. This process is shown in the italicized notations on the named flows and process box in Figure 2-2.

The mapping process shown in Figure 2-1 supports this notion of model-based software processes. The architecture, used as the control input, is an abstract model which is applied to the domain model to produce the generic design output. In Chapter 3 of this report, the mapping process will be refined using this model-based view.

To better understand the mapping process, the models used as its input, control, and output (FODA products, the OCA, and the Generic Design, respectively) are described in the next three sections. The resources (Domain Engineers and tools) are not further described in this report.

2.2 The Domain Model — FODA Products and Representations

The FODA method, as described in [Kang 90], describes three major products created during its Domain Modeling phase. They are:

1. The *Information Model*,⁷ which captures and defines the domain knowledge and data requirements that are essential in implementing applications in a domain.
2. The *Feature Model*, which captures the end user's understanding of both the general and specific capabilities of applications in a domain and describes:
 - a. the *context* of domain applications, depicting the variability of the users and environment for applications in a domain,
 - b. the needed *operations* and their attributes, and
 - c. *representation* variations.
3. The *Operational Model*,⁸ which identifies the functionality and behavior (both commonalities and differences) of applications in a domain. It provides the foundation upon which the software designer understands how to provide the features and make use of the data entities in the previous two models described.

[Krut 93] documents the use of a tool to capture the products of the FODA domain model. The tool is "001" from Hamilton Technologies, Inc., documented in [001SRM]. Since this tool was

7. Earlier reports used the term Entity-Relationship Model, but an ER model is only one format for an Information Model. A semantic data model or object model are alternative formats.

8. Earlier reports used the term Functional Model.

used to represent two of the three domain models, the pertinent 001 notation will be briefly explained in the following paragraph and examples of its use will be shown.

The basis of FODA Information Modeling is composed of two basic relationships: the **is-a** and **consists-of**.⁹ The is-a relationship is further refined by adding a third relationship, the **is-set-of**. Using these relationships, the entities used to describe the information content of a domain are described and organized. Such an organization can be shown using the 001 notation as described below.

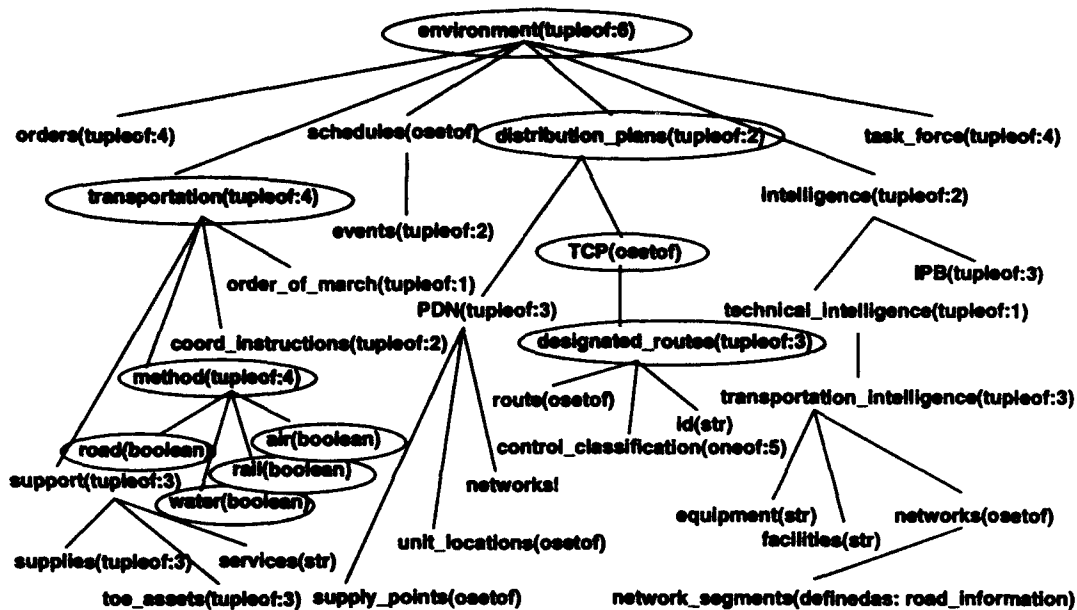


Figure 2-3: Example of an 001 Information Model

The 001 TMap (or TypeMap) provides a tree-like structure with each node corresponding to an object type. Figure 2-3, shown above, depicts an abbreviated version of the Information Model for the movement control domain. The TMap enables the modeling of the decomposition of objects using sets, arrays, trees, classification, reference, extension, and primitive types. These, in turn, map readily to the constructs and semantic notions of many programming languages. The concepts of generalization, aggregation, and attributes were transformed using the TupleOf, OSetOf, and OneOf abstract types within the 001 TMap syntax as follows:

- The decomposition of a parent object into different component parts (children types) was represented by the TupleOf abstract type, representing the semantics of the record construct. These component parts may be objects of the same type or objects of different types.

⁹ See Section 5.2 of [Kang 90] for a more detailed description of the usage of these constructs in FODA.

- The one-to-many, parent-child relationship was represented with the OSetOf abstract type. The OSetOf abstract type represents an ordered set of objects, containing zero or more objects of the same type. The OSetOf type can be implemented in any number of ways; lists and arrays are two examples.
- There exist entities (or objects) in which there are many possible children types yet, when an object instance is created, exactly one of the child types exists. These are represented by the OneOf abstract type. Languages that support variant structures readily implement these semantics.

To use an 001 model, the user (or tool using the internal representation) traverses the tree and uses the semantics corresponding to the 001 type at each node to understand the model and its contents. For example, the movement control *environment*, shown at the top of Figure 2-3, consists of six data aggregates, as represented by the TupleOf notation. One of these, the *distribution_plans* entity, consists of two items. The *TCP* (the Traffic Control Plan) is shown as an OSetOf of *designated_routes*, describing the notion of a collection or subset of the available roads which will be placed under specific control to regulate their usage. Under the *transportation* subtree and its *methods* branch, the four entities shown with the boolean options describe the notion used to the optional incorporation of *road*, *water*, *rail*, and *air* as transport mechanisms.

The 001 notation fully represents the semantic model concept of the FODA Information Model. The semantic model captures the bulk of the data requirements for the domain.

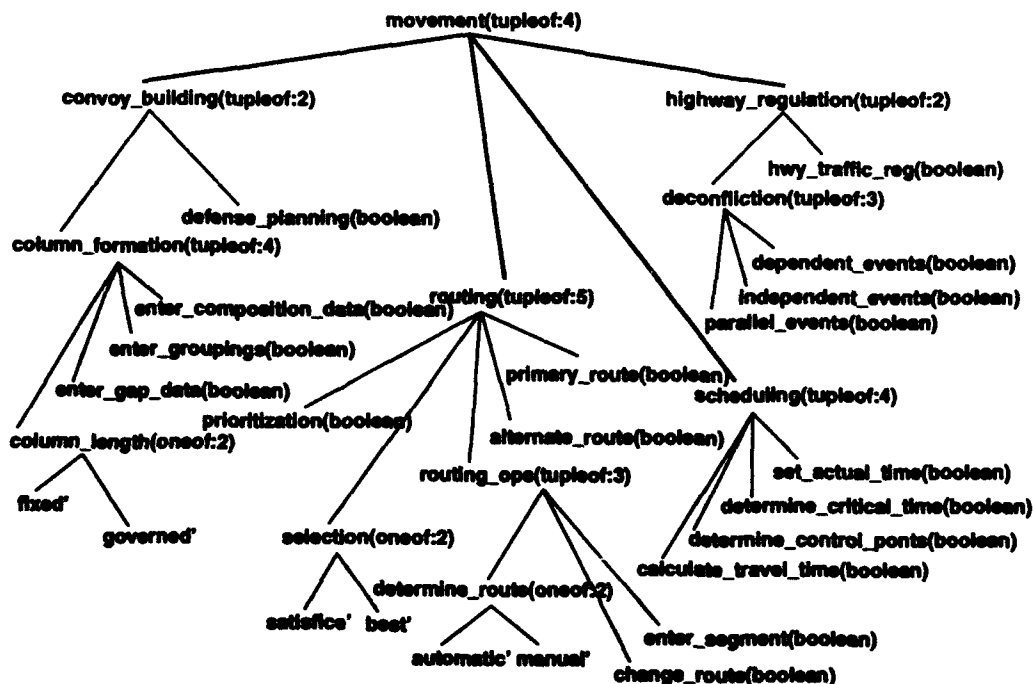


Figure 2-4: Example Representation of an 001 Features Model

The 001 TMap notation is also used to represent the features model. Figure 2-4 on the following page depicts a portion of the movement control features model, focusing on the features relevant to convoy planning. The features were identified and structured as optional, alternative, or mandatory as described in the FODA method, in the following ways:

- the *optional* features are modeled as leaf-node objects of type Boolean; this allows their later designation of their usage with True or False selection,
- the *alternative* features are modeled as a OneOf abstract type, which readily captures the notion of alternative,
- and *mandatory* features as a TupleOf abstract type.

Since a TMap follows the same tree-like structure as the baseline features diagram, the concept of "reachability" defined in the FODA report is maintained within a TMap.

The Operational Model is captured using any number of CASE tools which allow for the integrated definition of the functional and behavioral characteristics of applications software in a domain. The SEI reports cited previously describe the use of various tools to support this model.

Now that the pertinent FODA products have been described, it is time to discuss the architectural concepts needed to produce a generic design. The concepts are embodied in the OCA, which is the subject of the next section.

2.3 The Architecture — The OCA

In [Shaw 90], the term software architecture is defined as: *a software design at a level of abstraction that focuses on the patterns of systems organization that describes how functionality is partitioned and how those elements are interconnected*. There are two key parts to an overall organizational pattern, a *partitioning strategy* and a *coordination model*.

A **partitioning strategy** is the criteria used to *decompose large software problems into smaller subproblems and the allocation of those subproblems to software components that will solve them*.¹⁰ In the OCA, the partitioning strategy is realized by building blocks such as subsystem and surrogate structures and their components, and the executive. The **coordination model** is the *glue that binds separate activities into an ensemble* [Gelemter 92]. In the OCA, the coordination model is realized in rules and templates that determine how the building blocks interact with one another.

A key attribute of a good architecture is the separation of the coordination strategy or model (the flow of control through the software, or mission) from the providers of operations or services (the building blocks or components). This separation:

- allows a change in operation, or service provided (potentially due to a new piece of equipment or information), without requiring a change in the existing mission software, and

¹⁰. See [Abowd 93] and [USAF 93].

- allows a change in the mission without necessarily requiring a change in the service providers carrying out that mission.

Because of the clear separation between the partitioning strategy and coordination model, program families can be designed using the OCA that are highly modifiable with respect to broad classes of change. The next section presents an overview of the OCA, focusing on the partitioning strategy and coordination model embodied within it.

2.3.1 Overview of the OCA

The OCA¹¹ is an abstract model that provides an architectural pattern for the packaging of software. The architectural pattern embodied in the OCA allows software developers and reusers to distinguish the design and packaging of the service providers and the code elements that are mission oriented. The three architectural elements are called:

1. *objects* (service providing elements)
2. *controllers* (elements that embody a mission through use of objects)
3. *executives* (mission activator elements)

The architectural elements of the OCA and the components used to implement the elements are described further in the following paragraphs.

2.3.2 OCA Components

A controller and its associated objects are called a *subsystem*, a single product or family of products whose definition is wholly self-contained. The subsystem model is an essential element to understanding the OCA and is illustrated in Figure 2-5. The components of the subsystem model, Objects, Controllers, Imports, Exports, and Signatures, are described in the order given. Then a variant of the subsystem, the surrogate, is described. Finally, the role of the executive is discussed.

2.3.2.1 Objects

An object maintains state information about the behavior of a real-world or virtual entity. The kinds of real-world objects that can be modeled are things like engine parts, i.e., cams, pistons, rings, and lifters. On the other hand, an object can model a thing that is not physically realizable, such as a map that depicts the roads in an area. An object, when implemented, performs two important functions:

1. it provides services through internal subprograms, and
2. it maintains a readable state.

11. The OCA is based upon the Object Connection Update (OCU) paradigm developed by the Ada Simulation Validation Project (a funded SEI project from 1987 to 1989) sponsored by the U.S. Air Force Aviation Systems Command (ASC/YT) and described in [Lee 88]. Parts of this paradigm have been incorporated into the Structural Modeling process and framework described in [Abowd 93] and [USAF 93].

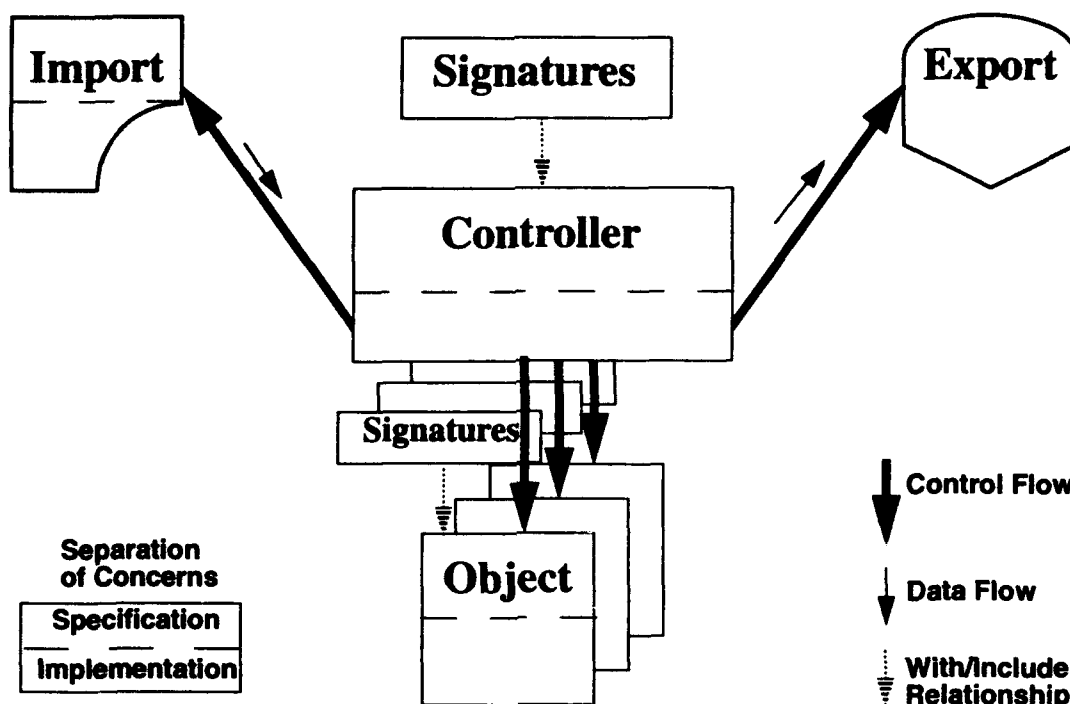


Figure 2-5: The OCA Subsystem Model¹²

The implementation of an object is abstracted through a *manager*. An object manager maintains a consistent procedural interface to the underlying representation of an object via recurring patterns for:

- the procedure names for its operations, and
- its usage of data passed as parameters.

Thus, no matter how the object changes the implementation of its operations or its internal representation, the operation names and their data inputs and outputs available through the manager should not change for a well-defined object.

2.3.2.2 Controllers

As an engine is an aggregate of its parts (the cams, pistons, rings and lifters), a controller aggregates objects to form a cohesive subsystem. A controller is the locus of information pertaining to the subsystem's mission, i.e., the specific activity or task with which the group of objects is charged (an engine provides rotational torque and power to the remaining drive train subsystems). A mission is a bounded activity within a single domain of expertise. The mission of a subsystem is captured in:

¹² In the figure, the line splitting the Controller, Object, and Import components is used to depict the separation of concerns between the specification of the callable interface and the implementation for each component.

- *what* objects are needed to perform a cohesive set of related operations at a level of abstraction above individual requirements (or low level features),
- *where* information inputs and outputs are located, and
- *when* low-level operations are invoked and in what order.

The controller is used to drive the subsystem, including the management of interactions between the objects within the subsystem and the usage of data elements via use of a set of explicit constructs for data transfer, the *import* and *export* structures. These two structures are described in the following two sections.

2.3.2.3 Imports

The import structure:

- is the locus of other subsystem state data needed by a subsystem to achieve its mission.
- collects state data from other subsystems' export areas needed to achieve a subsystem's mission.
- maintains separation of concerns between subsystems and their objects.

The import structure defines the interface for data input from the other subsystems. The controller accesses input data needed for object operations via this structure, thus the control flow from the controller to the import structure and the return data flow illustrated in Figure 2-5.

2.3.2.4 Exports

The export structure:

- is the locus of a subsystem's state data needed by other subsystems to achieve their missions.
- provides storage area for a subsystem for data reflecting the state of the subsystem's mission.
- allows access to required state data without requiring access to the object.

The export structure defines the data output interface for use by the other subsystems. The controller places the required results of invoked operations into this structure, as indicated by the control and data flows from the controller to the export structure shown in Figure 2-5.

2.3.2.5 Signatures

Signatures are a powerful mechanism for abstraction in the OCA; they are a formal representation of the interface to components (objects, subsystems, and surrogates) that provide services.¹³ Signatures play a major role in how the high degree of separation of control and data flow seen in the OCA is achievable in practice.

¹³. [Srinivas 91] describes the concept of signatures as a means of describing the key notions of domain entities as names and how the names form a vocabulary for describing a domain. Signatures were cited as the most fundamental of the three ingredients in the specification of a domain, as they are used within the axioms (formulas) and models in an algebraic specification. Signatures are also described as a useful notion when attempting to identify a component's reuse potential in [Zaremski 93].

For objects, Signatures consist of information about the details of the functionality, in particular the processing options, provided by the object's operations. Object signatures contain abstract names for the services (and their underlying algorithms) provided by the objects, and hide the mapping of the abstract service (and the selected name) to the implementation of the service. In this way, implementations may change and alternative algorithms may be selected by various users with minimal effort because users access the services through the logical names provided by the signatures rather than directly.

For subsystems, the Signatures include:

- Object Signatures information that must be visible to other subsystems, surrogates, and the executive,
- the names of all of the data items accessible individually and by aggregate via the subsystem controller operations,
- names for the internal state of the subsystem, used by the executive, to control overall system flow.

The signatures for surrogates are equivalent in content to those for subsystems.

Subsystem signatures allow the use of a subsystem's operations without explicit reference to (or knowledge of) the underlying objects comprising it, the specific features they provide, or even the names that have been chosen to represent the services at the object level. A signature for a subsystem may provide varying degrees of abstraction; the most trivial (and least abstract) subsystem signature contains the union of the signatures of each object in the subsystem. More sophisticated subsystem signatures provide higher-level services by mapping simple names onto:

- services provided by objects,
- specific instantiation of services provided by objects, such as an invocation of an object's services with a specific set of parameters, or
- complex sequences of the object-provided services.

For example, an object might encapsulate a database that represents a terrain map of a particular area. Its signature might include facilities for returning the height above sea level at a given location, each facility with different accuracy and computational characteristics. This signature would remain stable even if the database were replaced with one of less resolution that might require the object to extrapolate among the heights or nearby coordinate locations.

Additionally, this database object might be part of a subsystem that computes the as-travelled distance between two points. The signature for the subsystem might represent a service that returns the distance, given the starting point and destination. This signature may, unknown to those outside the subsystem, map to a series of object operations, such as deriving a point-to-point routing along roads between the given points or calculating distances.

Signatures provide a conceptual link from the executive to lower levels of functionality that is independent of the implementation of that functionality. Figure 2-6 illustrates this linkage. The dependency links depict the incorporation or usage of a Signatures structure by another structure and are shown by the solid arrows. The dashed arrows in the opposite direction reflect the visibility of the Signatures names to the structure using them. With the transitive inheritance of the Object Signatures (via the Subsystem Signatures) into the executive, the executive now has sufficient visibility to invoke object operations in the desired manner, thus the conceptual link between them. Only those parts of the Object Signatures needed by the executive are required to be passed along by the Subsystem Signatures. These are the names of those low-level processing options that must be visible to the executive (for any number of reasons). The names provide a sufficient description to express the semantics of the option without violating the coordination model, where rules explicitly prohibit invocation of object operations from within the executive directly.

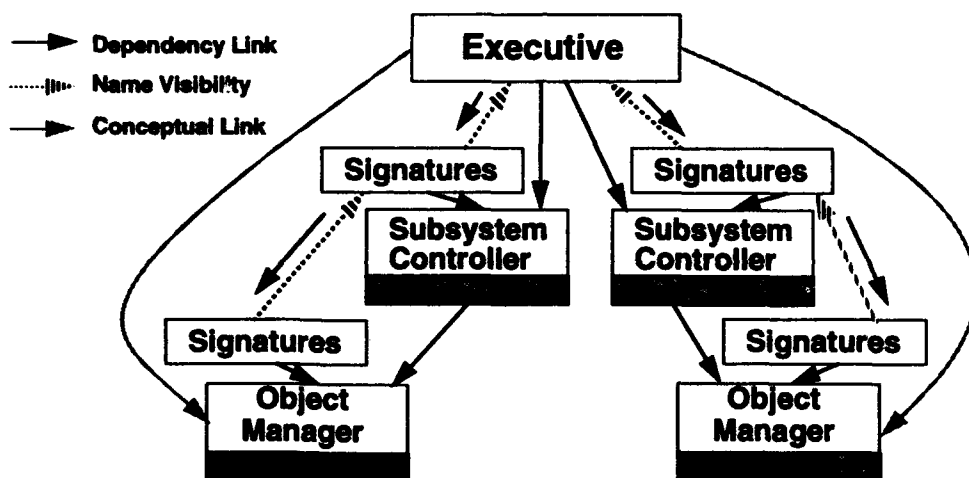


Figure 2-6: Signatures as Conceptual Links to Object Functionality

Sections 5.3 and 5.5 describe specific examples of the uses of signatures for objects and subsystems, respectively.

2.3.2.6 Surrogates

An important variation of the subsystem, the *surrogate*, is used to aggregate information about physical or logical devices with which the application is to interface. These devices include:

- file systems or databases,
- other computers via hardware devices and software protocols, and
- human users via devices ranging from dumb terminals and keyboards through bitmapped displays with pointing devices and sophisticated graphical support packages.

The word surrogate is defined in [AHD 85] as: *one that takes the place of another; substitute*. The most important goal of the surrogate structure is to provide a sufficiently abstract notion of the logical device as to allow for the substitution of different physical devices or implementations (i.e., different operating systems and file structures). If a device is sufficiently well abstracted, then exploiting its capabilities in applications is only a matter of:

- characterizing the physical attributes of the actual device to be used in a specific application, and
- characterizing the data to be handled by the surrogate for the application.

The key differences between subsystems and surrogates lie in two major areas:

1. Surrogates are intended to be domain or product line independent, at least to the level of the format of the data types provided within the application. This domain/product independence also extends to the consistency of the names for its operations, versus the domain specific operation names used within subsystems. The notion of names for subsystem and surrogate operations is discussed further in Sections 5.4 and 5.6.
2. A surrogate structure may be incomplete. For example, an input-only device will not import any data and an output-only device will not export any data to be used by other subsystems. It is expected that all subsystems will require both an import and export structure to be fully specified.

The overall structure of surrogates is equivalent to that for subsystems. They contain controllers, imports, exports, and a signatures structure at the controller level. The objects underlying surrogates, the *handlers* and *transforms*, are structures that provide the necessary interfaces to the device(s) and the functionality to implement data exchanges between the application and the device(s). The internal structuring of surrogates into handlers and transforms is discussed further in Section 5.8.

2.3.2.7 Executives

The executive provides the operating environment for the subsystems within the application and, in most cases, is the arbitrator over conflicts between processes competing for time and access to shared resources. The executive monitors and controls time for subsystems and monitors interfaces to external entities such as hardware devices, other computers, and humans, i.e., application users through the surrogates. The role of the executive in the overall flow of control in the OCA is clearly described in the next section.

2.3.3 Flow of Control and Data in the OCA

When the OCA is used consistently as the basis for implementing the operational aspects of a domain's features, the resulting subsystems and surrogates can be readily combined into complete applications with an executive. The executive provides the appropriate level of control over the application via the subsystem controllers. Yet, because of the use of the Signatures structure, the executive has no visibility to the data needed in the application, thus achieving an important separation of concerns in the resulting software. The flow of data and control is illustrated in Figure 2-7.

The solid arrows show the flow of control within the system. Except for the handling of error conditions and the surrogates providing feedback, control always flows downward. Subsystems do not make calls to other subsystems or their objects. Calls to objects are made only by the appropriate subsystem controller. This standardizes the control flow which increases the understandability and maintainability of the software. The objects and subsystems also have mechanisms for error handling and recovery which will be discussed in detail in a subsequent report.

The dashed arrows indicate the flow of data between the subsystems. Data placed in a subsystem's export structure is available for use by other subsystems via their respective import structures. Export structures have no knowledge of where their exported data are used. It is the responsibility of the import structure, when implemented, to know where to get the data necessary to provide all required inputs for the subsystem's operations.

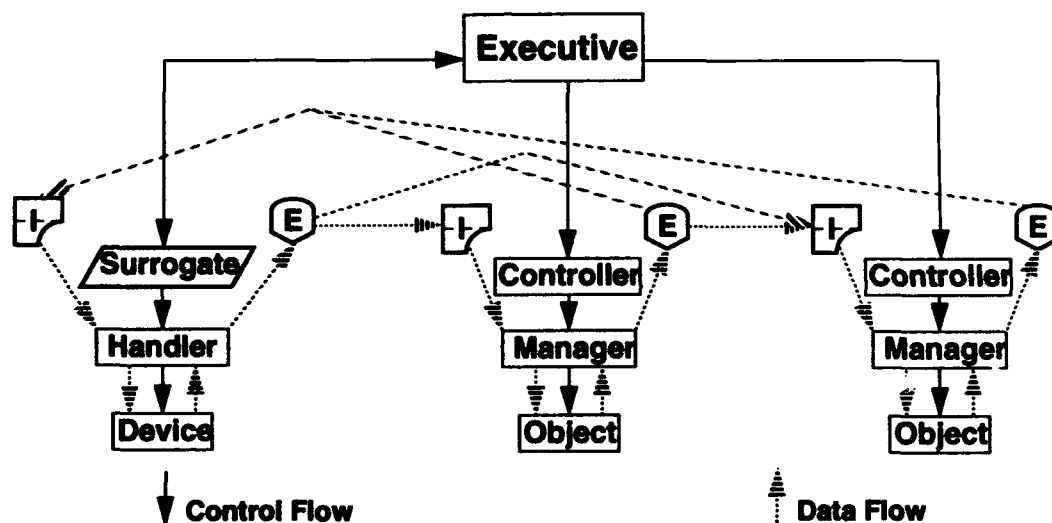


Figure 2-7: Overall Flow of Data and Control in the OCA

In summary, the OCA is an architecture that allows one to specify a generic design for software systems using objects managed by controllers under the direct supervision of an executive where:

- objects model the behavior of real-world (or virtual) components and maintain state.
- controllers aggregate objects and manage connections between them based upon a reaction strategy.
- executives manage the subsystem and surrogate controllers, time, and the application state to provide acceptable response to stimuli.

A discussion of some of the benefits the OCA provides in application development and maintenance is given in Section 3.4.

2.4 The Generic Design

The resulting output of performing the mapping process is the generic design. This design may take many forms, depending upon the architecture used as a control. The important concept is that the generic design is a domain dependent instance of the selected architecture. It should always be possible to recognize a generic design as an instance of architecture X, if that architecture has a well-understood partitioning strategy and coordination model as described in Section 2.3.

For example, Figure 2-8 depicts the top-level structure for the convoy planner prototype application developed by the authors during the initial execution of the mapping process. The use of the OCA is readily apparent in the designation of subsystems and surrogates, and in the uniform flow of control between them and the executive. Appendices G and H further reflect the use of the OCA, as they capture instances of use of the specifications forms and templates as represented in Appendices D and E, respectively.

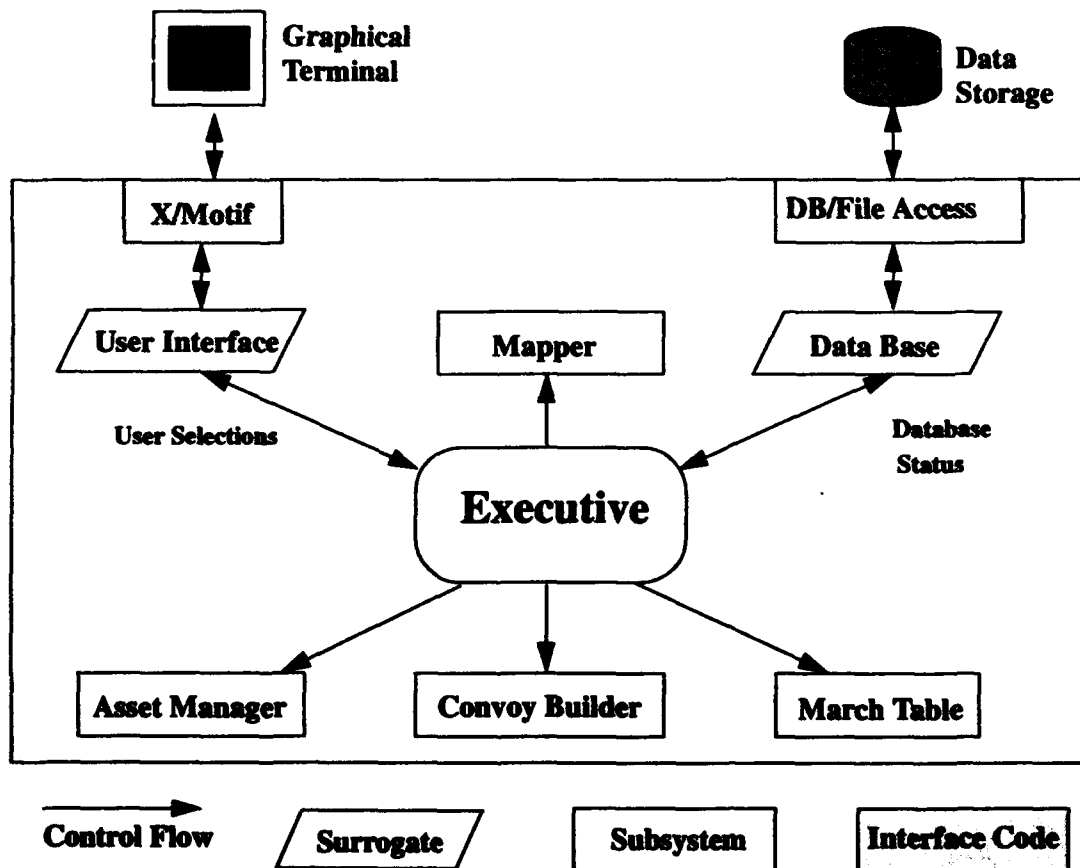


Figure 2-8: Convoy Planner Generic Design

In describing the usage of models in software engineering and the role of the domain model architecture and generic design, the context for understanding the mapping process is complete. The next chapter provides an overview of the mapping process, including its applicability and limitations.

3 Overview of the Mapping Process

Previously, the mapping process has been described using a SADT view of a process with input and output products as various kinds of models.¹⁴ Now that the models pertinent to the mapping process have been described, that view can be refined to focus on the use of those models within processes that together describe the entire mapping process.

3.1 Partitioning the Process

The mapping process has steps that apply to both the Domain Engineering and Application Engineering processes as described in [Withey 94]. Figure 3-1 shows an expanded view of the mapping process with three subprocesses illustrated:

1. **Domain Design**, which takes a Domain Model as input and applies a Partitioning Strategy from an Architecture as a control model to produce a Generic Design.
2. **Domain Implementation**, which takes a Generic Design as input and applies Code Templates and Design Rules from an Architecture as a control model to produce the Supporting Components (for the Generic Design).
3. **Application Development**, which takes a set of Components and a System Specification as input and applies Rules and Code Templates to produce the Application Code.

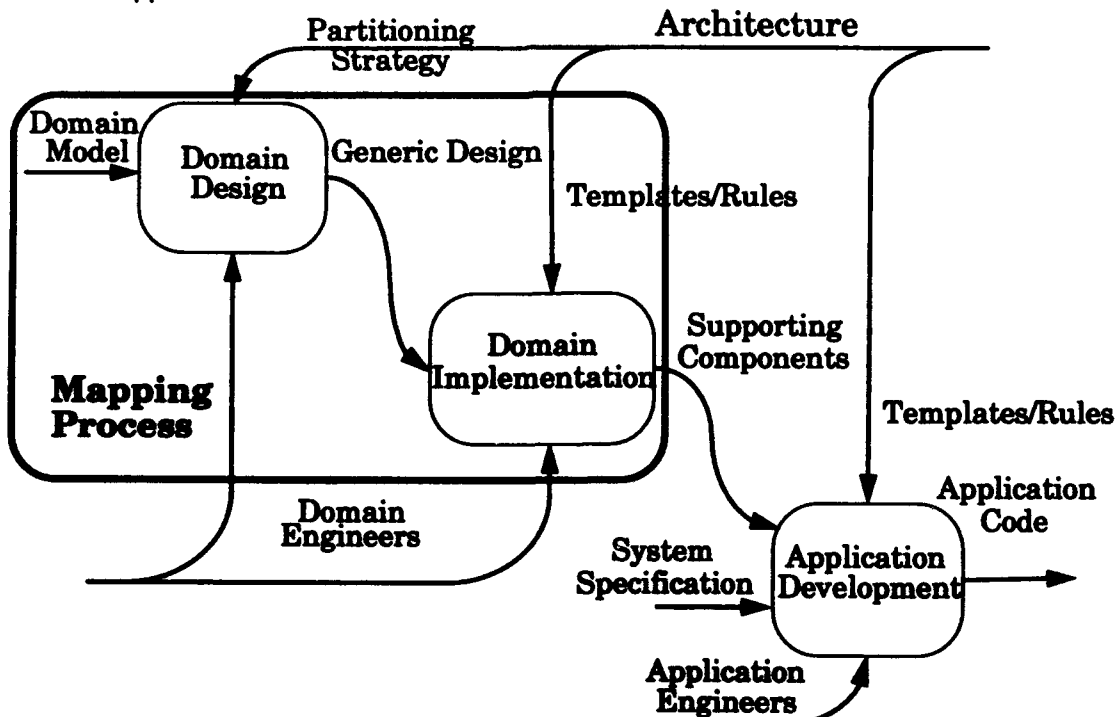


Figure 3-1: Decomposition of the Mapping Process

¹⁴ See Figure 2-1 on page 7 and Figure 2-2 on page 9.

The Domain Design and Domain Implementation processes together form the mapping process which is the primary focus of this paper. However, these processes do not present a complete picture in that there is no process pertaining to the use of the resulting concrete models (the design and components) in the development of application products. The Application Development process is intended to complete the process of migrating domain knowledge into delivered products in a systematic manner.

Chapters 4 through 6 define a series of steps, segmented into three processes as summarized above. The goal of the Domain Design process is to collect the information needed to develop the package structures that implement the OCA. This information is collected onto a set of forms (shown in Appendix D) that will be the major inputs for the Domain Implementation process. In Domain Implementation, a set of code units (Ada package templates are shown in Appendix E) which implement each subsystem, object, or surrogate defined during the Domain Design process. Chapters 4 and 5 describes these two processes.

The Application Development process takes the user through some portions of the process of using the completed or partial components to develop an application within the domain. Chapter 6 describes this process.

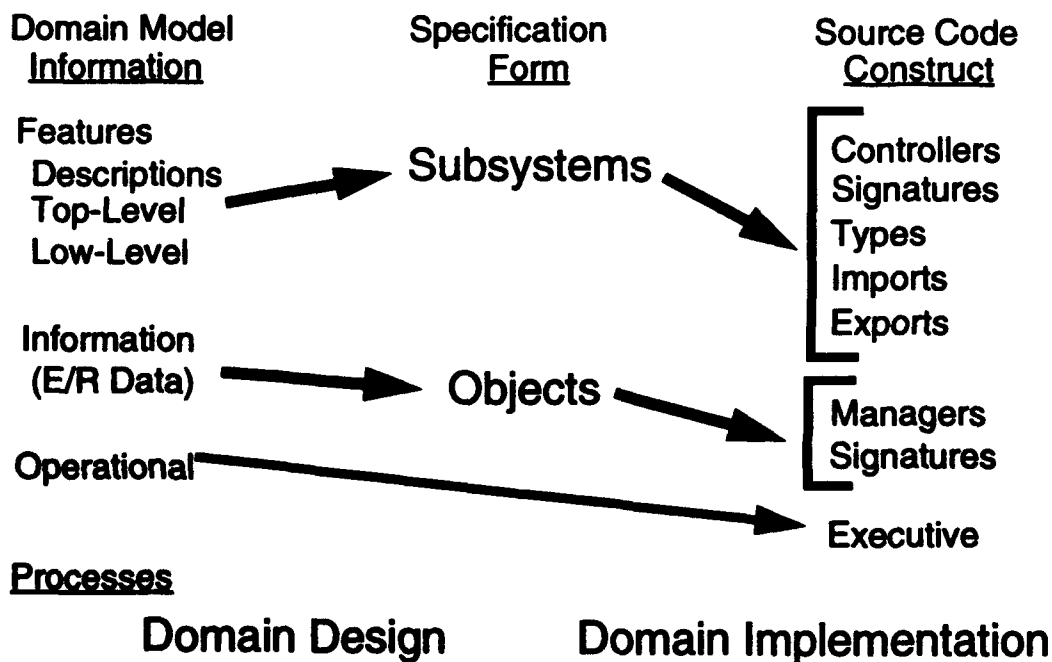


Figure 3-2: Migration of Domain Data to Specifications and Code

Figure 3-2 illustrates the overall flow of the mapping process, showing the transitions from the domain model to code via the use of the specification forms. Note how, during the Domain Design process, the specification forms gather information from diverse sources via the multiple models that collectively denote the domain and then how that information is

repartitioned into multiple code units during the Domain Implementation process. This is similar to how requirements documents are written for systems, gathering information from diverse sources into a single document (or set of documents), and how those documents are used. This similarity and others within this development process are the focus of the next section.

3.2 Viewing the Process as a Normal S/W Development Process

The mapping process is analogous to the generic software development process of producing:

1) a specification, then 2), a design that meets the specification, and finally 3), code that fits within the design. The analogy holds because:

1. the Domain Design process provides specifications for subsystems, surrogates, and objects to fit within a generic design. This process links together portions of domain models products into descriptions via forms suitable for further refinement. The specification forms serve as the requirements for the software entities to be developed.
2. the Domain Implementation process provides detailed designs for subsystems and surrogates that satisfy the specifications from the previous phase. These designs are instances of the OCA subsystem model, using the components described in Section 2.3. Although the production of code via the use of templates is the focus of this process, the code is not tied to any specific application but is meant to be usable by all applications requiring the capabilities of the subsystem or surrogate.
3. the Application Development process provides code for an application that conforms to the OCA. This process binds together the subsystems and surrogates, completing those portions of their code components left incomplete from the Domain Implementation process, according to the requirements of the application in terms of hardware (specific devices), software (specific capabilities or features), and desired performance characteristics. This process also involves construction of the executive, which has its own template and guidelines for its completion.

This analogy is important because it links the mapping process to the conventional software development process as practiced in most organizations. Hence, the mapping process is not costly to implement in their overall process. This realization within an overall software development should ease the organizations's transition to use of this process. Transition planning for this process is discussed extensively in [Withey 94].

The next section describes how the mapping process and its use of the OCA applies to the problems of developing reusable software.

3.3 Use of the OCA in the Development of Reusable Software

The OCA is a model for organizing and implementing the structure of an application. Its focus is on the high level problems of control and data flow within systems, and not on the implementation of low-level functionality. The data flow is controlled by the import and export structures via the controllers; the main flow of control is handled by the executive.

The executive is designed to be a control structure for execution of an application using the OCA's subsystem model. As such, it does not have any visibility to data items being used within the application and, therefore, needs access to only the Signatures and Controllers for the subsystems and surrogates that compose the application.

The executive is built around the consistent usage of layered case statements. These layers provide a regularity of form within the code that results in an executive with a highly readable, understandable, and maintainable structure. The fact that the subsystem is used as the outermost layer of the executive's internal structure provides the developer with an easy mechanism to add or remove a subsystem; just add a new layer of template and complete it for a new subsystem, or delete the entire case layer and references to its operations in other layers to remove it. Appendix C.3 further describes the layers of the executive.

The OCA also provides applicable guidance in the structure of code for subsystems and objects. However, the OCA does not enforce any style for components below these abstractions. This does not mean that there is no appropriate guidance on how to implement components at lower levels. Many object oriented design (OOD) methods exist that can be used to create highly flexible and, hence, reusable components that would be applicable for use within the OCA. Section 5.9 provides further insight into a five-layered scheme for subsystems and their objects using abstract data type (ADT) packages and instances of them.

3.4 Benefits from Reusing OCA Structures

There are two major benefits to the use of the OCA in developing a generic design:

1. consistency of form within applications, and
2. separation of control flow and data flow.

Each of these benefits is discussed in the following paragraphs.

3.4.1 Consistency of Form Within Applications

The use of templates for code and the rigidity of the OCA in terms of code structure may be foreign to many software developers who are used to developing in an individual or project-specific style. However, once developers become familiar with the usage of the OCA, then they will find that it provides a solution for most design problems. Think about how much time designers have used answering the question "How do I want *this* system to look?" The normal answer is some ad-hoc decomposition whose justification only exists in the developer's mind and will probably never be understood by future maintainers. The OCA forces the developer to ask a different question: "How do I design this system using the OCA?" The OCA

provides a look for a system, and this look is consistent for all systems using the OCA as an design basis. This means maintainers can make important assumptions about *how* a system does what it does, and can focus on understanding *what* it does, vastly simplifying the maintainer's workload.

3.4.2 Separation of Control Flow and Data Flow

The OCA is designed to separate the flow of control (what subprograms get called in what order) from the flow of data (where are the required inputs and where do the results get put). Control flow is implemented in:

- the Executive
- the Subsystem and Surrogate Controllers, with support from the Signatures packages
- the Object Managers and the packages they use in their implementation

On the other hand, data flow is implemented in:

- the System Engineering Units and Subsystems Types packages which declare the types
- the Export packages which declare the variables/objects of the types
- the Import packages which map data type usage to available Export variables/objects
- finally as parameters to Object subprograms and their underlying implementation

This clear separation of concerns produces an application with control and data flow responsibilities being relegated to distinct sections of the code structure, again aiding developers and maintainers in understanding the system and where changes of various kinds go within the code modules comprising it. Thus, the OCA provides a consistent system architecture that makes the post deployment software support process a much more manageable task and makes for longer lived systems with lower costs.

3.5 Limitations of the Mapping Process

At the highest level, the mapping process is highly dependent upon the abstract models that are applied to create the resulting concrete models. The process would need some modification to incorporate the use of different domain analysis products. The OCA is only one instance of an abstract architectural model that could be applied to produce a generic design. The mapping process could be very different depending upon the kind of architecture chosen because the later steps depend heavily upon use of design rules and code templates associated with a specific architectural model. The OCA has a well-defined component structure, in terms of kinds of components and their interactions. This leads to a number of code templates to support the components and corresponding interaction rules. A more loosely defined architecture may have few supporting templates and rules. More work needs to be done to determine the applicability of this process when applied to different architectures.

The mapping process does not fully specify all of the decision making logic needed to implement the process in an organization or for a particular project or application. Synthesis steps, where disparate pieces of information are combined together, are generally complete in that the steps state what information is being combined and the form of the result is well understood. The analysis steps, however, are more heuristic in that they provide insight into the intent of the step and a characterization of the desired output, but no precise rules can be given for making specific decisions.

Use of the OCA limits the use of certain programming paradigms in the implementation of subsystems and their objects. The internals of objects must be passive in that they cannot invoke functionality except within their own scope. It is the role of the subsystem controller to coordinate the interaction of its objects; the executive coordinates objects in different subsystems through the controllers. In object-oriented programming (OOP) terms, methods or operations cannot call other methods/operations. This is to reinforce the notions that objects are only service providers and that they can take no active role in the application's control flow beyond their limited scope.

3.6 A Roadmap for the Details of the Mapping Process

To this point, the focus of this report has been on a general description of the mapping process, a context for its applicability in Model-Based Software Engineering, and an understanding of the kinds of models needed to describe and implement the process. With this done, the reader has a sufficient understanding of the goals and intents of the mapping process and the details of the process can be given. Chapter 3 begins the description of the process at a detailed level.

The use of domain analysis products to develop designs and supporting software components prior to their use in individual applications is commonly referred to as Domain Engineering. This chapter covers the steps in the core of the mapping process, which is applicable to Domain Engineering. The processes and steps in this chapter are meant to be performed apart from Application Engineering, which embodies individual product development.

Chapters 4 and 5 delineate the steps to support the Domain Design and Domain Implementation processes, respectively, as seen in Figure 3-1 and introduced in Section 3.1. The FODA domain model provides the inputs and the OCA's specification forms, code templates, and design rules serve as the controls for the domain engineers who produce the generic design and its supporting components.

Figure 3-3 on page 29 presents the OCA structures and the process steps discussed in Chapters 4 through 6. After selecting the capabilities to be mapped (described in Section 4.1), Figure 3-3 illustrates the use of specification forms that map the relevant information from the domain model using steps in Sections 4.2 - 4.4. The forms then map onto the set of code components seen in Figure 3-3 by performing the steps in Sections 5.1 - 5.9 and in Chapter 6. Each form or code component used (starting with the templates in Appendices D and E) is marked with the step or steps relevant to its use.

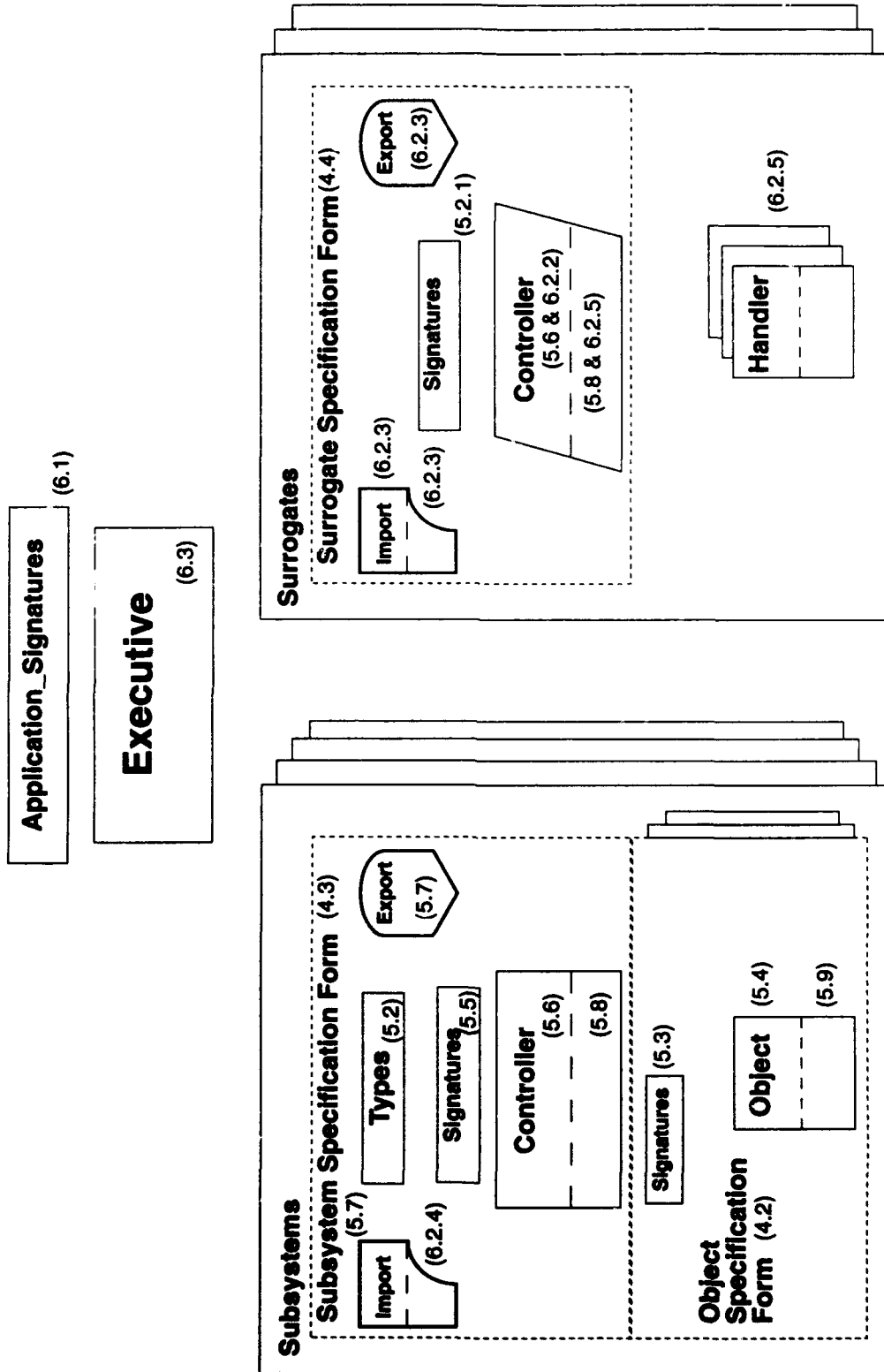


Figure 3-3: Mapping Design Elements to Process Steps

4 The Domain Design Process

The *Domain Design Process* for subsystems and objects that fit within a domain-specific generic design consists of 4 steps, shown with a summary of their actions and products in Table 4-1.

These steps map various portions of FODA concrete models onto appropriate sections of specifications forms for subsystems and objects (see Appendix D, Sections D.1 and D.2, respectively, for examples of these forms). This mapping is summarized in Table 4-2.

The goal of the specification forms is to provide the software developer with access to the information needed to implement functionality. They are designed with the assumption that a domain model is in place. Therefore, except for information that can readily be transcribed in a succinct way, they provide references to information within the various products of a FODA domain model. The forms provide pointers to information elements about data, capabilities, and behaviors from the Information, Features, and Operational Models, respectively.

The steps listed in the sections below are given in the order in which they should be performed. The only notable exception is that the creation of subsystem and surrogate specifications, described in Sections 4.3 and 4.4, can be performed in either order or concurrently.

Step	Action	Product
1. Select Features from Domain	Identify desired features from features model, i.e., operations, context, and representation.	List of desired features
2. Create Object Specifications 1. Identify Objects	Identify data items maintaining state or requiring explicit control.	Initialized Object Form, Entity List
2. Derive Object Operations and Inputs/Outputs	Analyze features model for operation variations based on alternatives or context and shown in operational model.	Completed Object Form
3. Create Subsystem Specifications	Group together objects that work together, correlated to set of related features in features model.	Completed Subsystem Form
4. Create Surrogate Specifications for Devices	Determine external interfaces for applications and determine their control and data characteristics.	Initialized Surrogate Form

Table 4-1: Summary of the Domain Design Process

Domain Model Information	Subsystem Specification Form/Section	Object Specification Form/Section
Features		
Descriptions	Description Requirements Exceptions	Requirements Exceptions
Top-Level	Features	N/A
Low-Level	N/A	Features
Information (E/R Data)	Objects Imports Exports	Name Description
Operational	Imports Exports	Imports Exports Exceptions

Table 4-2: Mapping Domain Model Constructs to Specification Forms

The description of each step in this specification processes is given at a level above the clerical work of completing the specific forms. Each step will be described as follows:

- its *name* will be given (in the title for the section),
- a summary of the *action(s)* taken in the step,
- the *input(s)* used in the performance of the step, and
- the *product* resulting from completion of the step and how it contributes to the process.

Appendix A contains the details of the work to be completed at each step of the Domain Design process concerning the use of the forms. Its sections are numbered to correspond to the equivalent step in the following sections.

Step 0 — Establish an Overall Goal for the Mapping Process

Before beginning the mapping process, it is important to know what the major goal of the process is, because various alternatives exist:

1. One can pursue a limited set of features that map readily to a core set of capabilities that are to be used in a product or as a domain demonstration (as done in the movement control example). However, the exclusion of features (those not selected) may impede the ultimate reusability of the software if and when those features are later desired.

2. One can include all features of major capabilities into the process, which can lead to the most robust and reusable design possible. The development of such a design and its components would be a very difficult task because of the complexity of implementing and integrating the use of a potentially highly diverse set of features and underlying objects.

These two extremes, building in only what you need (as in 1.), and building in everything you could ever want (as in 2.), have different success criteria and cost versus benefit tradeoffs, as briefly described above. In an organization transitioning towards MBSE, the model may be to start more with a "build in what you need" goal as a start to build a reasonable set of core components, transitioning to "build in everything" as your organization and product line knowledge matures.

Each application of the mapping process probably involves some combination of these alternative process goals at varying levels of engineering decision making.

4.1 Select Features from Domain Model

The selection of features involves the analysis of the feature and operational models to determine:

1. what *features* are required or desired in the software to be implemented, and
2. what *functionality* and *entities* are necessary to deliver those features.

The feature model provides the primary input for deriving the requirements for the software to be implemented because they capture the essence of user needs and desires for the software. As described in Section 4.3.2 of [Cohen 92], there are three distinct groups or classes of features for the movement control domain (and, one can assume, for most domains). These three classes of features and their effect on the software specification(s) and framework are:

1. *Operations*. Those features that describe the functional characteristics of the domain; the services that a system must provide.

The operational features are essential throughout the process of creating the object and subsystem specifications. Many of the steps in the process make direct references to information in the feature model.

2. *Context*. Those features that describe the overall mission or usage patterns of a system; the description of the class(es) of users for a system.

The context features serve two major purposes in this process:

- They may drive the selection of various operational features, i.e., omission of alternatives or options, based upon the specific context to be implemented or, conversely, inclusion of multiple alternatives and options based upon a desire to support a wide range of potential contexts.
- They will manifest themselves in variations of control flow in controllers and executives as derived from the control information contained in the operational model.

3. **Representation.** Those features that describe how information is viewed by the user or produced for another system; what sorts of input and output capabilities are available.

The representation features will be a significant driver in the kind and capabilities of the physical and logical devices needed to support the resulting application, and thus the surrogates needed to interface with those devices. Representation features can also drive the aggregation of data and their structures as defined in the objects.

Without knowledge of the desired features, it is not possible to determine the kinds of data needed in the software or the necessary structures or constraints to be placed on that data.

The result of this step is a record (in no specific form) of the selected features. This record (list, highlighted features diagram, or other media) is used throughout the remainder of the Domain Design process.

4.2 Create Object Specifications

Now that a number of features have been selected for inclusion in a generic design, the next step in developing the design is to understand how the features/capabilities are to be provided. The service provider abstraction in the OCA is the object. Thus, analysis to locate, understand, and specify the objects needed to implement features is the most appropriate step to perform at this point.

4.2.1 Identify Objects

Those entities that are required for the selected features and functionality need to be created. Look first for entities that are first and second entities above primitive entities. The primitive entities usually embody data items at an elemental level of representation, i.e., numeric values, string data, etc. The entities above these leaves generally provide useful software abstractions for the domain.

Object Form

Name	Description
------	-------------

The place to look for objects to be supported is within the information model. In FODA, the information model is created to *support analysis and understanding of the domain problems and to derive and structure domain objects used in the application.*¹⁵ The primitive entities are the leaf items in the Information Model, while the object's abstractions are found at various levels in the tree-based information hierarchy, as described in Section 2.2.

Higher-level objects may be constructed from combinations of lower-level objects. From the movement control domain, the concept of a road map involves knowledge of:

1. *locations* - names of places, their positions in terms of some coordinate system, and other identifying or useful data.

¹⁵ See [Kang 90], Section 5.2.3.

2. *segments* - connections between locations, their names, and other useful information, in particular, their length.
3. *routes* - a series of segments that defines a path between two or more locations of interest.
4. the *map* itself - a complete collection of the locations and segments in a selected geographical region.

It is not necessary to provide an object for each entity in the information model. There is an important criterion to be used to determine whether or not an entity in an information model should be mapped onto an object. In the map description given above, the locations, segments, and routes are easily implemented as simple information aggregates, such as records and arrays in the Ada language. Conversely, the map concept involves the use of a complex structure due to the need to maintain the interrelationships between segments and locations, i.e., the segments know about which locations they connect, but locations may be connected to arbitrarily many locations via multiple segments. The generalized criterion can be stated as follows:

- If a data entity can be implemented as a simple data structure with no extra functionality required to support the abstraction,
then leave it as an entity, noting its existence on an Entity List for incorporation into a System Engineering Units or *_Types* package (defined in Sections 5.1 and 5.2, respectively), as appropriate.

Use a single Entity List for all of the objects to be specified via this process.

- If the entity consists of a non-trivial relation between data items where explicit functionality is required to maintain that relationship,
then the entity should be allocated to an explicit object. This object's internal state will support the needed relationship via use of abstract data type (ADT) packages in the implementation of the object.

Each simple entity has been captured on the Entity List, for later specification in a data type package. Each entity selected to become an object will be specified using a Object Specification Form, shown in Appendix D.2. Further specification of the object, deriving its needed operations, is performed in the next step, described in the following section.

4.2.2 Derive Object Operations and Input/Outputs

The operations on the selected objects are derived from two sources:

1. the selected features and feature combinations in the features model (from the record developed during Step 1), and
2. by determining the transformations needed to support the data flows specified in the operational model.

Object Form

Requirements
Features
Imports/Exports
Exceptions

The transformations and the flexibilities required by pertinent features are synthesized into the object's methods or subprograms when implemented in the selected programming language.

The different classes of FODA features may be handled as follows:

- All *mandatory* features that are immediate descendants of the selected feature are entered on the form.
- *Alternative* features that are considered applicable or desirable to be incorporated are entered on the form. Choose at least one alternative feature: otherwise, the parent feature cannot be correctly mapped into an implementation. Be sure to note that the feature is an alternative.
- *Optional* features that are considered applicable or desirable to be incorporated are entered on the form, noting that the features are optional so that the developer can make the use of that feature available for easy addition or deletion.

It is desirable to attempt to enumerate all of the relevant features on the Object Form. One can separate between:

- features supported directly in the implemented object (building what you need), or
- features supported via multiple instances of the object with various selections of feature availability and underlying implementations (building in everything), and
- those features not supported by the object or any version.

However, it will be very important to be able to trace the implemented object (and versions) to a general specification of the range of capabilities desired.

The Object Specification Forms for the allocated objects are complete at this time. They are used to specify the Object Manager and Signatures components, which are completed within various steps of the Domain Implementation process. Figure 3-2 on page 24 shows the migration of domain information to these OCA components. Table 4-2 on page 32 refines this migration by detailing the usage of the various sections of the Object Specification Form.

4.3 Create the Subsystem Specifications

As discussed in Section 2.3, a subsystem is an aggregation of objects. At this point, it is appropriate to synthesize a grouping of the objects that will be working together to perform a given mission. These objects should correlate strongly to some set of features in the features model which are collected under a single mid to high level feature. Depending on the scope of the domain, a subsystem should correspond to some feature in the feature model where the objects to be aggregated support the needed subfeatures for the subsystem encapsulating the selected parent feature.

Subsystem Form

Name
Description
Requirements
Imports/Exports
Exceptions

For each input or output needed or provided by a subsystem's objects, determine the appropriate source (for imports) or destination (for exports) subsystem using the operational model. This step may be deferred until the application is further defined because the

applicable input source may not be defined until all subsystems and surrogates have been delineated. However, it is imperative that all required inputs be noted so that system completeness can be determined, that is that all inputs for subsystems have at least one source from some other subsystem or surrogate.

The completed Subsystem Specification Form, shown in Appendix D.1, is produced by performing this step. Figure 3-2 on page 24 shows the mapping of domain information to the OCA subsystem components at an overall level. Table 4-2 on page 32 details the data captured within the various sections of the Subsystem Specification Form.

4.4 Create a Surrogate Specification for Each Logical/Physical Device

This step specifies an abstract view of a logical or physical device that is to be part of an application within a domain. These devices include text or graphics terminals with keyboards, pointing mechanisms, disk drives, and communications mechanisms. The use of such devices may be an integral part of the total application. For example, any program that interacts with a user and uses previously stored data needs two surrogates:

Surrogate Form

Name
Description
Type
I/O Connections
Imports/Exports
Exceptions

1. one for the user interaction device, i.e., the terminal/keyboard, and
2. another for some form of persistent data storage.

The isolation and abstraction of such devices is an important step in the creation of a consistent framework for use by multiple applications in a domain.

There are two important kinds of information needed for the analysis to specify the abstraction to be embodied within a surrogate:

1. the *control* characteristics of the device to be abstracted. These characteristics will be identified using the following terms:
 - a. a *monitor* device - a device which, upon receiving an appropriate stimulus from an external source, causes the application to *receive* control signals and/or data to which the application should respond. Such devices are nominally thought of as *input* devices.
 - b. a *control* device - a device which, upon receiving an appropriate stimulus from the application, causes the device to *send* control signals and/or data to an external destination. Such devices are nominally thought of as *output* devices.
 - c. a device that exhibits both of the above behaviors; a device that performs both input and output functions.

Also, consider whether the device behaves in a synchronous or asynchronous manner.

2. the *data* characteristics of the device, in terms of both:

- a. the *physical* characteristics of the initial input or final output, in terms of size (of the device's buffer area, if any) and layout.
- b. the *logical* characteristics of the final input or initial output, i.e., its form.

The implementation details of how the control signals and data are transformed from formats usable within the application to the format needed by the device being abstracted, and vice versa, are what is hidden by the packaging under the surrogate specification.

The identification of surrogates can come from several different areas in the domain model and the products that comprise it. The representation features in the feature model can identify the needs for operations that must be provided by user interface devices with varying degrees of capability, text, and/or graphics. The need for persistent data, shown in the operational model, requires a surrogate to handle data storage and retrieval operations.

The completed Surrogate Specification Form, shown in Appendix D.3, is the result of this step. The mapping of needed information from the domain model is very similar to that for subsystems, as described in the previous step.

5 The Domain Implementation Process

The steps in the previous section describe the Domain Design process, which derives the specifications for the three logical abstractions within the OCA: objects, subsystems, and surrogates. The following section will describe the *Domain Implementation* process of mapping the information in those forms onto Ada templates for the component structures in the OCA implementation.

Step	Action	Product
1. Identify/Create Engineering Units	Identify low-level units of measure or base data types from Information Model and Context features and create or select packages to handle them.	Reuse of applicable data types across subsystem boundaries
2. Create Subsystem <i>_Types</i>	Identify data types needed for object operations visible to subsystem externals and create definitions.	Encapsulation of subsystem data types
3. Create Object Signatures	Create namespace for selecting object operations alternatives from the Features To Be Supported items.	Encapsulation of object operational variations
4. Create Manager specification	Create object operation profiles using consistent naming and with needed parameters (data/features).	Specification of Object abstraction
5. Create Subsystem/Surrogate Signatures	Export Object feature information and create namespace to document data entities manipulated by the subsystem's objects.	Specification of Subsystem entities and operational variations
6. Create Subsystem/Surrogate Controller specification	Create subsystem operation profiles using consistent naming and Signature parameters.	Specification of Subsystem abstraction
7. Create Subsystem/Surrogate Import/Export packages	Create support for acquiring needed inputs for operations and making outputs visible to others.	Encapsulation of data interface functionality
8. Create Subsystem/Surrogate Controller package body	Create sequences of object operations to achieve subsystem requirements, importing and exporting data as needed following operational model.	Implementation of Subsystem internals
9. Create Manager implementation	Implement object operations using low-level components/ADTs, etc. following operational model.	Implementation of Object internals

Table 5-1: Summary of the Design Implementation Process

The Domain Implementation process of mapping specifications for the subsystems and objects onto the OCA and its Ada code constructs consist of nine steps, shown with a summary of their actions and products in Table 5-1. The steps in the process are given in an order that allows the developer to complete a code unit, usually by filling out a template, and then compile the resulting source code it into a program library. Subsequent code units developed in this process can make use of these previous results.

The Ada language incorporates the notion of a *package* construct as a group of logically related type, object, and subprogram declarations. Other languages, such as C++ or Modula-2 have an equivalent notion and corresponding construct, such as the *class* or *module*. In addition, these languages also provide the equivalent of the separation of concern between specification of operations and their implementation seen in the Ada package *specification* and *body*, as illustrated in the steps shown in Table 5-1. Modula-2¹⁶ defines the separation in terms of distinct *definition* and *implementation* modules. C++¹⁷ clearly defines (in a less formal manner) the class interface (or header, placed in a .h file) and implementation (placed in a .c file). Whenever these steps refer to the terms package specification or package body, one can refer to use of the equivalent structures in other programming languages.

The first six steps of the Domain Implementation process involve the bottom-up construction of code unit specifications up to the level of the subsystem controller. The last three steps are a top-down refinement process of developing supporting package specifications and bodies, i.e., the import and export areas, and the bodies of the package for the subsystem controller and the object manager (implementation details discussed in the appropriate section and/or corresponding appendix section).

Steps 3, 4, and 9 are performed for each Object Specification Form derived from the previous process. Steps 2, 5, 6 and 8 are performed for each Subsystem or Surrogate Form. Much of the actual work of surrogate development is deferred until the application structure is completely defined and will be described further in Chapter 6.

These nine steps map information on specifications forms for subsystems and objects onto various constructs within the resulting code structures. This mapping is summarized in Table 5-2 on the following page.

The description of each step in this process is given at a level above the clerical work of completing the specific template(s). Appendix A.2 contains the details of the work to be completed at each step concerning the use of the Ada templates. Its sections are numbered to correspond to the equivalent step in the following sections. As in Chapter 4, each step will be described in terms of its name, actions, inputs, and product. Each step results in a completed OCA component as a product, except for Import bodies and surrogate components which are completed during the Application Development process described in Chapter 6.

16. See [Wirth 85] for further details on the Modula-2 language.

17. See [Stroustrup 91] for further details on the C++ language.

Specification Form/Section	Source Code Construct
Subsystem	
Requirements	Controller
Features	Controller Signatures
Objects	Signatures Types
Imports	Imports
Exports	Exports
Exceptions	Signatures
Object	
Requirements	Manager (procedures)
Features	Signatures
Imports	Manager (procedure parameters)
Exports	Manager (procedure parameters)
Exceptions	Manager (exceptions declared within the Manager and propagated by procedures)

Table 5-2: Mapping from Specification Forms to Code Constructs

5.1 Identify or Create Applicable System Engineering Units Package(s)

The goal of developing a System Engineering Units (SEU) package is to gather together the information about the basic elements of data that are commonly used as parts of many other abstractions in software systems. The inputs are the Entity List, which specifies the need for various data items, and the Information Model, which specifies the characteristics for the data.

Many of the low-level parts of data items within typical software systems are probably derivable from a relatively small number of data types whose usage spans many domains. Various units of measure, such as feet or meters depending on which measurement system the software is to support, and the operations needed to support their use, are prime examples.

There is no standard name for this package because it may not be prudent to put all of the shared or common types into a single package. In the movement control example, type declarations of this nature were built into three packages:

1. **Measurement_Types** - a package of basic units of measure such as length, weight, and speed, which are available in both metric and English units and

scales, and which provides a flexible mechanism for using and changing between the units.

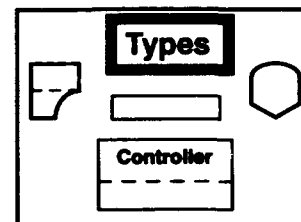
2. **Vehicle_Types** - a package which encapsulates the highly variant structure of the hierarchy of information that can be stored as part of the many different views a vehicle may have within the Army command structure.
3. **Default** - a package which provides useful constants for various instances of **Vehicle_Types**.

The identification of data types that should become part of this package or set of packages (depending on the number and layering of data types needed) can be identified by examining the Entity List, developed during the Identify Objects step in Section 4.2.1, for those data items that are used by **multiple** subsystems. Developers should attempt to reuse packages that provide the required data types wherever possible. Some good examples are the **Basic_Data_Types** and the many mathematical packages available from the CAMP project (see [McNicholl 88]). Otherwise, the developers must implement them directly. [Gautier 90] contains a suitable set of guidelines for development of Ada packages with specific rules for generics and a strong focus on reusability.

The result of this step is the selection of reusable data type packages to be used throughout the OCA components. These packages specify many of the data types to be transferred between the import and export packages for subsystems and surrogates and the parameters used by object operations.

5.2 Create Subsystem **_Types** Package

The process and inputs for creating this package are similar to that just described for creating the System Engineering Units or **Common_Types** package(s). The key result is to provide a suitable type definition for those entities that will be used with a **single** subsystem.

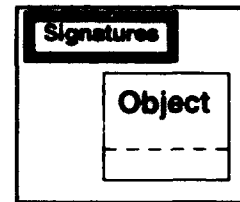


Allocate all of the remaining data items in the Entity List to the appropriate subsystem's **"_Types"** package and create a suitable type declaration for each item, referring to the Information Model for the needed data characteristics.

Notice that all of the data structures that will be passed between subsystems and surrogates have been defined in these two steps. Those data items that are used by multiple subsystems are defined first in various SEU packages. Then, those items that are used only in one subsystem are defined in specific **_Types** packages. Surrogates are an exception in that they need access to the data types of all subsystems to which they are to provide their services. Thus, the import and export packages and the controller package body for a surrogate will incorporate type information from each subsystem **_Types** package as needed to implement the needed operations for the surrogate.

5.3 Create Object Signatures Package

The Signatures package at the object level of the OCA captures the use of features that imply a selection from alternative algorithms or other information used to control the execution of an operation, e.g. control information. As an example, consider the operation of selecting a route between two points (following the logical map structure described in Section 4.2.1).



The movement control domain model in [Cohen 92] describes two alternative subfeatures for the route selection feature, best and satisfice.¹⁸ Computer science has two terms that are synonymous with these notions, optimal and heuristic. A best (or optimal) solution should guarantee the most accurate results possible, at the cost of high (or perhaps prohibitive) computational time. Conversely, a satisfice (or heuristic) solution may provide a result that is less than optimal yet is satisfactory for the intended purpose, and has the property of being solvable in an appropriate time period. The selection of the best or satisfice feature has nothing to do with the data (in this case the points selected as start and end points and any others to be visited along the way). The feature selection has to do only with the algorithm used to produce the desired results, i.e., control of the application. This is the intent of the use of features in FODA, the selection of appropriate functionality according to the user's desires.

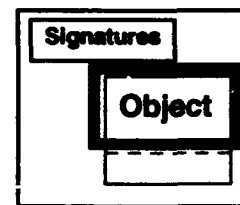
The Object Signatures package captures the flexibility of an Object Manager's functionality. The Signatures package creates the namespace for identifying the existence of features, keeping separate from the code that creates and implements the operations that satisfy those features.

Building the Object Signatures packages is a straightforward process. For each named feature from the Object Specification Form, create a corresponding name within the Object Signatures package. If an object has no features, i.e., alternative implementation calls or optional processing, then no Signatures package is required.

Creation of Object operations begins with the next step, as described below.

5.4 Create Object Manager Package Specification

The intent of the Object Manager is to provide a standard mechanism for invoking the operations needed to provide the services for the physical/logical object that is to become a part of a system. Use of a standard mechanism provides two major benefits:



1. It allows the use of a predefined set of operations/names that give objects a consistent set of operational semantics. This enables objects that implement

¹⁸ The word satisfice is defined as *to decide on and pursue a course of action that will satisfy the minimum requirements necessary to achieve a particular goal* [OED 87]. This word and its definition are attributable to Nobel laureate Dr. Herb Simon, who has used it in many contexts, including finding various classes of solutions to combinatorial problems ([Simon 81], Chapter 5, p. 138.)

even highly different abstractions to be common building blocks for multiple applications due to the external similarities.

The development of this consistent namespace for object (and eventually subsystem) operations involves an analysis that is beyond the scope of this report (see Limitations in Section 3.5).

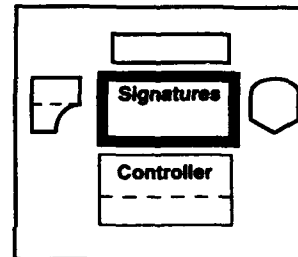
2. It results in a great increase in the understandability and maintainability of the subsystem operations that utilize the object's (via the manager) operations in their implementations.

The need for parameters for the operations is defined by the lists of *Imports* and *Exports* in the Specification Form for the object. Using the Operational Model to provide the precise needs for inputs and outputs for each operation, fill in the parameter profiles for each operation specified in the code template. If an object handles multiple entities, then each predefined subprogram will be overloaded to handle each of the entities to be stored and/or manipulated by the Object via Manager calls. Also, for each type of object feature, add a parameter of mode 'in', using a default parameter value if a particular feature is nominally used at invocation.

Using the information on the Object Specification Form, verify that each mandatory feature is allocated to one (or more) subprograms and that specified optional or alternative features are similarly supported via use of feature parameters in the appropriate subprograms. Finally, ensure that there exists a version of the applicable subprograms for each entity allocated to the Object Manager.

5.5 Create Subsystem/Surrogate Signatures Package

As discussed previously in Section 5.3, the Signatures package is a mechanism for creating a namespace to be used by the executive to achieve precise control of the subsystem. The executive achieves this control via the passing of control-related information from the executive to subsystems and their underlying objects during procedure invocation. The process to be followed for creating a Signatures package is similar to that described earlier for Objects. The features named on the Subsystem Form are mapped onto operation names or to features names to be passed down to invoke alternative or optional sets of Object operations.



For the Subsystem Signatures, there is the additional responsibility of incorporating any Object Signatures packages and their information into the Subsystem package and *reexporting*¹⁹ them for use in the executive. Reexporting makes the various type names and enumeration literals from the associated Object Signatures packages directly visible at the subsystem level. The effect of reexporting the types and literals from the Object Signature packages is to minimize the need for other subsystems and surrogates or the executive to

¹⁹. See [Bardin 87a] and [Bardin 87b] for an explanation of the concept of reexporting and for an extended example of its application, respectively.

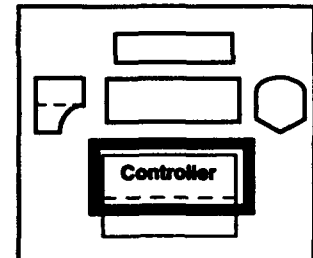
require visibility to the Object Signatures, i.e., using the *with* (as in Ada) or *include* (as in C++) syntax to reference the signature information. Reexporting simplifies the usage of the subsystem by centralizing all of the Signatures information into a single entity, the Subsystem Signatures package.

A User Interface (UI) Surrogate Signatures package provides a good example of the use of signatures in the OCA. The UI is a focal point for control information in that the user's input in terms of menu selections, button clicks, etc., must be transferred in a form usable by the application's executive. Similarly, errors that the user must be informed of need to be transformed into device-usable formats. Thus, the UI Surrogate Signatures is a complex structure that, in many cases, requires a great deal of knowledge about the application as a whole. Its *Status_Value* is a variant record structure that contains the equivalent of the selected menu option, keystroke, etc., that are to be processed by the application as a user command. All subsystems whose operations are invoked by the user via the UI surrogate must become part of the surrogate's namespace. Also, the information needed to construct appropriate error messages must be in the Signatures package.

A surrogate depends upon information from subsystems it is to support; therefore, it will not be possible to completely define a surrogate until all of the subsystems that will become part of an application are selected and defined, e.g., a complete surrogate is bound to a specific instance (at the subsystem composition level) of an application architecture. Consequently, the addition or removal of a subsystem to an application will (in all probability) affect one or more of the application surrogates. This does not mean that a surrogate is application-specific; just that there is some degree of coupling between subsystems and the surrogates. This coupling is well defined and is described in Section 6.2 where the surrogates are completed.

5.6 Create Subsystem/Surrogate Controller Package Specification

The Subsystem Controller provides a uniform interface to the underlying capabilities of the subsystem. Again, the use of a predefined namespace for the subsystem provides for consistent names across subsystems, just as for objects as discussed in Section 5.4. The procedures declared in the Controllers use the entities declared in the Signatures package as parameters. The parameters will be used in the Controller body to determine which Object operation(s) need to be invoked to fulfill the required functionality.

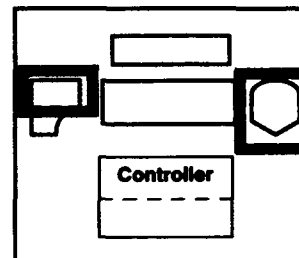


For surrogates, the process is essentially equivalent to that for subsystems. Again, some amount of work must be deferred until the application is defined, since each of the two device control procedures, *Application_To_Device* (for control surrogates) and *Device_To_Application* (for monitor surrogates), is overloaded for each functional subsystem that is part of the application architecture. Also, because for some subsystem operations we need to distinguish between differing kinds of information relating to the same entity, we may

need to use an alternative form of the *Device_To_Application* procedure. This second overloaded form allows an incomplete yet useful amount of entity-related information to be transferred into the application versus a complete data item. This alternative is useful in passing search key information into the application for use by a subsystem in searching for and locating a desired piece of information in a large information aggregate so that a complete record can be returned, hence the use of the term *Key* in the *Data_Kind* declaration seen in the template in Appendix E.3.

5.7 Create Subsystem/Surrogate Import and Export Packages

As discussed in Sections 2.3.2.3 and 2.3.2.4, the Import and Export packages provide the mechanism for bringing data into and getting it out of the subsystem or surrogate; there is no difference in their implementation, other than when in this process they can be completed. The subsystems can essentially be completed at this point, whereas the completion of the surrogates must be deferred until the application context is further defined.



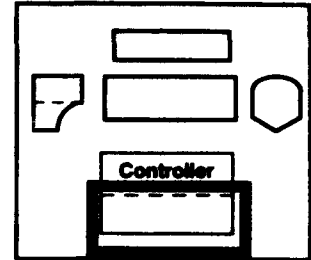
The Export package is completed first. The need for a data item to be exported is defined by the Export and Destination information in the Subsystem or Surrogate form. The Export package creates a visible reference to data values of use to other parts of the application. The content of the declared data items is determined via calls to subsystem/surrogate operations. There are two basic methods for exporting data within the OCA:

1. One can simply supply declared variables which are directly visible within the package.
2. One can supply functions that return an applicable value. The functions hide any visibility to the variables themselves and can be used to implement indirect and/or mutually exclusive access to the data.

The Import package is completed in two parts. The specification or declaration part creates the function name (one for each data type listed in the Imports section of the Subsystem Form). The function will be invoked by the Controller to gain access to the data value specified by the return type (one for each data type required by one or more subsystem operations. The body supplies a reference to a data value of the required type), either directly by naming a variable created in another subsystem's Export package, or indirectly by calling the function supplied by the Export package. In either case, the knowledge of where a data item comes from is hidden at a level where it is easily modified and has minimal effect to the remainder of the application code. The body part is completed in Section 6.2.4 for subsystems. For surrogates, completion of the import and export packages is deferred until Section 6.2.3.

5.8 Create Subsystem/Surrogate Controller Package Body

The Subsystem Controller body is where the binding occurs between the objects that provide the services and the needed imports and exports needed to make the services perform work. Thus, the recurring theme in implementing a Controller body will be centered about the following sequence:



1. Take in the entity (and feature(s)) parameters eventually to be passed in by the call from the Executive, defined in the operational and feature models, respectively.
2. Based upon the given parameters, select the appropriate Object operation to call.
3. Call the selected operation, using appropriate Import functions to satisfy the input parameters and Export variables to receive output results.
4. If no exception is raised, ensure the state of the subsystem is indicated as Normal, or handle any propagated exception by either issuing other Object calls to clean up the Object state or setting the Internal_State to a meaningful Status_Type for examination and action by the Executive when the Signal function is called.

The steps for completing a Surrogate Controller body are less clear due to the differing nature of what a surrogate is to accomplish in terms of the multiple of different capabilities presented by various devices. Thus, these steps are much less precise than those given in other sections of this report, but they will still provide some guidance to the implementer of the surrogate.

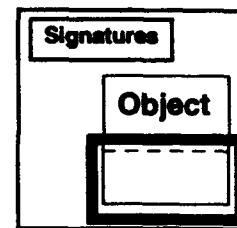
1. Plan to develop a surrogate in either one of two ways:
 - a. One style is to encapsulate the device characteristics into a single Handler package using the information on device characteristics and buffer size given on the Surrogate Specification Form and develop a Transforms package for each subsystem and its data entities to be handled by the device. This approach is recommended for devices that have a single set of fairly immutable characteristics where those characteristics are best located in a central location. This approach was used in the User Interface Surrogate (see Section F.1.1 for a discussion of data interfaces used in the user interface).
 - b. An alternate approach, suitable for disk systems whose file formats are highly flexible, is to develop a Handler and Transforms combination package for each subsystem. This approach acknowledges the fact that there is a single logical device (with potentially asynchronous behavior) that should be encapsulated in a controller, but the flexibility of the device precludes the need for a single Handler package, i.e., each file type or set of related files to be used is implemented as a separate package.

2. The namespace for device Handler and Transforms package is less rigid than the namespace used for subsystem controllers and object manager operations. It is important to develop a consistent namespace for use within a single surrogate, but the names need not be consistent across different surrogates. For example, the notion of Read, Write, Open and Close operations are highly useful when thinking about file systems, whereas Get and Put or Send and Receive operations are more applicable to other kinds of devices.

Also, since the surrogate will need knowledge of the specific application to be complete, the remaining portion of the surrogate development must be deferred until a specific application (in terms of subsystems/features) is to be developed.

5.9 Create Object Manager Package Body

The Object Manager body is where the binding occurs between the specific algorithms and/or abstract data types (ADTs) to be used to implement operations and the data types and constructs of the OCA. Therefore, the bulk of the completion of its internals is left to the implementer using any design approach or method that produces components that will integrate into the OCA subject to the limitations described in Section 3.5.



From this point on, the package implementer has to 'with' in any additional packages containing the algorithms and ADT operations desired to complete the construction of the operation subprograms in accordance with the requirements/features allocated to the object. This includes appropriate performance or system features such as constrained memory usage.

A subsystem controller is designed to be the top level of a hierarchy of code units supporting the application executive. Each level in the hierarchy provides specific resources or encapsulating lower level resources. Figure 5-1 illustrates the implementation of this hierarchy with an example, again drawn from the movement control domain.

The Subsystem Implementation Model depicts the contents of the Object Manager and the relationships between the components that form its contents and to the subsystem controlling the object. The bottom two layers of the hierarchy, Basic Operations and Data Types, provide ADT services (usually in the form of Ada generics) and type-specific declarations and services, respectively, to the next two layers, Utilities and Objects. The Utilities layers provide an area where generic objects can be formulated and those generics instantiated as needed prior to use within an Object Manager at the Objects level. The Object layer is where the use of Utilities and Data Types are combined into meaningful abstractions for use by the Subsystem.

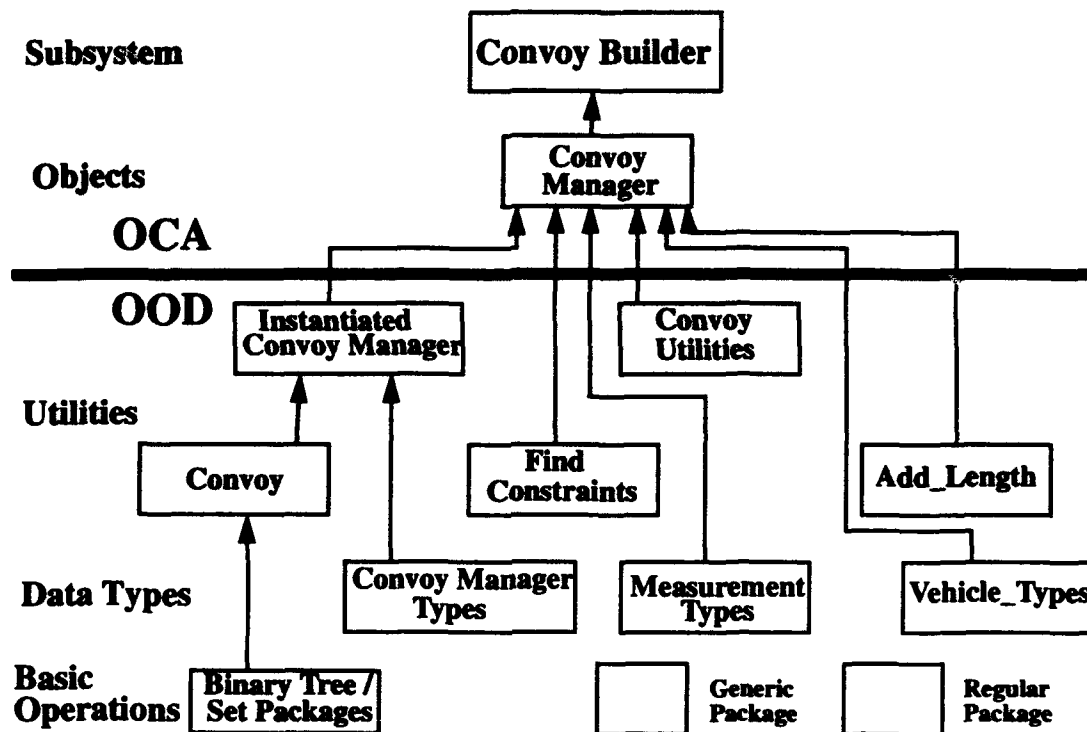


Figure 5-1: Subsystem Implementation Model

The structure of objects is less regular than that of subsystems or surrogates as defined by the OCA. The Subsystem Implementation Model shown here describes the kind of relationships and dependencies that can exist at the Object level. The Utilities, Data Types, and Basic Operations layers are an attempt to capture these relationship in an object-oriented manner. Again, the purpose of an object is to model an important entity in an application. It is not possible to rigorously define the implementation of an object because of the wide variance in what objects are needed in domains and how they are characterized in software. The Subsystem Implementation Model of the OCA provides guidance on the kinds of software components that can be used to implement domain objects in software.

6 Application Development Using a Generic Design

The previous two chapters described processes for taking information from the domain model and transforming it into design and code units (at varying degrees of completeness) that are to become the building blocks for the development of actual applications in the domain. The following sections describe the *Application Development* process, which completes any unfinished blocks and assembles them into a cohesive application. This process consists of three major steps, shown, along with a summary of their actions and products, in Table 6-1.

Step	Action	Product
1. Create Application Signatures	Determine needed subsystems and surrogates. Document operation namespace. Determine executive statespace.	Context for Application
2. Complete		
1. Surrogate Signatures	Determine data items to be handled. Determine control feedback needs to executive Determine error handling information needs.	Namespace to describe device capabilities
2. Surrogate Controller specification	Create surrogate operation profiles using given names and data item namespace.	Specification of Surrogate abstraction
3. Surrogate Import/Export	Use operational model to determine required external inputs/outputs.	Encapsulation of data interface functionality
4. Subsystem Import body	Complete mapping of sources for required inputs.	Isolation of data source location
5. Surrogate Controller body	Create sequence of transforms for handler operations.	Implementation of Surrogate internals
3. Complete Executive Template	Determine overall flow of control via operational model.	Top-Level Control of Application

Table 6-1: Summary of the Application Development Process

As described in the introductory material to Chapter 5, whenever these steps refer to the terms package specification or package body, one can refer to use of the equivalent structures in other programming languages.

6.1 Create an Application Signatures Package

The Application Signatures package is the top-level namespace for the application to be built. This namespace is most essential to the executive, but as discussed earlier, is needed by

various package bodies to complete their implementation. This package is where the various subsystems and surrogates are identified together as an Application Aggregate. Also, other important declarations useful to the executive and various surrogates, in particular the UI surrogate (if needed), are made here.

Three different kinds of declarations are made in this package:

1. naming of the subsystems and surrogates that constitute the application,
2. naming the subsystem/object callable operations,
3. naming the statespace for the application,

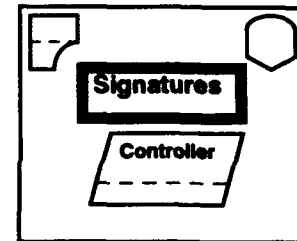
6.2 Complete Packages Making Use of Application Signatures

Now that the application is defined, via the namespace and the underlying references to the needed subsystems, the parts of subsystems and surrogates that were deferred from the processes and steps listed in Chapter 3 can now be completed. The completion of these packages is done in the paragraphs below, performed in the given order.

6.2.1 Complete Surrogate Signatures Package

Now that the subsystems to be used are selected, the surrogates that provide the interfaces for data to be processed by the subsystems can be completely defined.

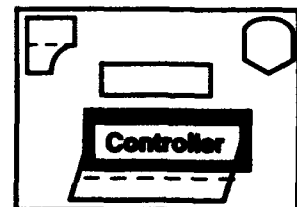
The entity namespace is the list of identifiers for those data types handled by the surrogate. Simply create an appropriate enumeration literal for each data type to be used.



For most surrogates, the `Status_Value` enumeration type was previously defined by the error information returned by the device. For the UI surrogate, the `Status_Value` is the list of valid operations whose implementation requires the services of one or more subsystems. For each subsystem with features, create a variant record whose discriminant is based upon the subsystem's `Entity_Type`, and then, for entities whose use is associated with a feature, create a variant part with a field to hold the identifier for the selected feature type. The `Status_Type` is then completed using the format given in the Surrogate Signatures Code template. Also, for UI surrogates, the `Error_Return_Type` template is completed, filling in a variant part arm for each subsystem to include its `Error_Type` information.

6.2.2 Complete Surrogate Controller Package Specification

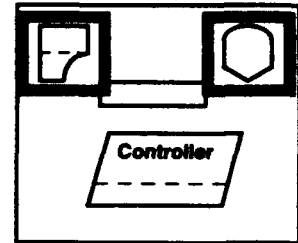
Completion of the *Surrogate Controller* code template (specification), started in Section 5.6, can now begin. Depending on whether or not a single `Entity_Type` was created in the surrogate's Signatures package or the use of the subsystem Signatures packages is required, the number of required subprograms can vary significantly.



The two subprograms, `Device_To_Application` (for allowing the device to make inputs visible to the application by placing value into the Export region) and `Application_To_Device` (for sending control information and data) are overloaded once for each `Entity_Type` from the surrogate or subsystem Signatures, as appropriate. Again, depending upon whether or not keys are required (as described in Section 5.6), use of the `Device_To_Application` subprogram template with the additional `Kind` parameter may be desired.

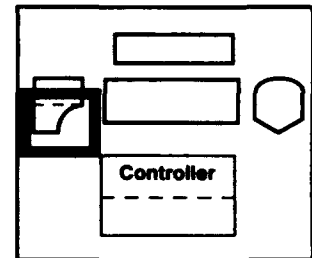
6.2.3 Complete Surrogate Import/Export Packages

This process is equivalent to that given in Section 5.7 for completing the subsystem Import/Export packages.



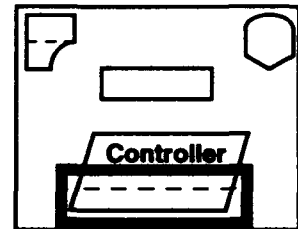
6.2.4 Complete Subsystem Import Package Body

With the completion of the Application Signatures package, the Import package body can be completed for those function bodies that required the use of the `Source` parameter, as described in Section 5.7. Additionally, data values coupled to external inputs via the surrogate Export packages can also be defined.



6.2.5 Complete Surrogate Controller Package Body

Completing the Surrogate Controller body is simply a matter of following through with the style of Controller selected in Section 5.8. The use of outside generic packages for creating the Transform package(s) to be the service providers for the surrogate controller is on an as-needed user-defined basis.



6.3 Complete the Executive Template

Finally, the construction of the application executive can begin. A code template should be used to assist in the implementation of the executive, such as seen in Appendix E.15. Within the executive, there are at least three major runtime phases, illustrated by the `Application_State` type in the `Application_Signatures` package. The initialization phase brings the application up to a point where it can receive inputs and produce results under the specified circumstances. Any operations needed to initialize the application must be called before the `Program_State` is set to `Steady`.

Within the main program loop, the `UI Signal` function acts as the driver for the application, i.e., the user selects operations to be performed and the application responds to those selections. The selections are returned in the form of the `Status_Type` record from the UI surrogate which has embedded in it:

- the identity of the *subsystem* of primary concern,
- the *operation* (translated into one of the standard operation names), and
- any *entity* and/or *feature* information needed to control the subsystems and underlying objects to be invoked by the executive as a result of the selection.

The selection may invoke a sequence of operations involving multiple subsystems or surrogates.

The executive main loop is organized internally as a set of nested Ada case statements. By using the pieces of the selection record in a consistent way, the executive can, in turn, be organized in a consistent way using the following scheme:

1. The first level of decomposition is at the *subsystem* level. Even though multiple subsystems play a role in most user operations, each user operation is relegated to a subsystem or surrogate which has primary responsibility, usually because it is the main service provider or the destination of the ultimate result.
2. The second level is the *operation* name, e.g. Construct, Destruct, and Fetch.
3. The third level is the *entity* name/identifier. This designates the type of data to be received, manipulated, and or returned.
4. The lowest level is the *feature* identifier. As described in Section 5.3, this value, if provided, designates a certain class of processing to be used in obtaining the desired result.

Thus, completing the main body of the executive control loop is a process of filling out the case statement template for each subsystem, operation, entity, and feature selection, as appropriate, and using the Operational model and other information to determine the appropriate sequence of subsystem operations for each user selection.

Finally, as in initialization, the executive can call a series of finalization operations on subsystems to ensure adequate storage of important results. The finalization section is reached only after some operation has changed the Program_State of the executive to the Finalize value. After completion of the finalization operation, the program terminates normally.

7 Conclusions and Future Directions

7.1 Conclusions

The mapping process described in this report provides a mechanism for using the information in the various models derived from exercising the FODA method on a domain. The process provides for development of the specification and implementation for software to be reused in a family of programs within that domain. The process is practical and provides precise guidance where applicable, yet is flexible enough to be used across a wide variety of application domains.

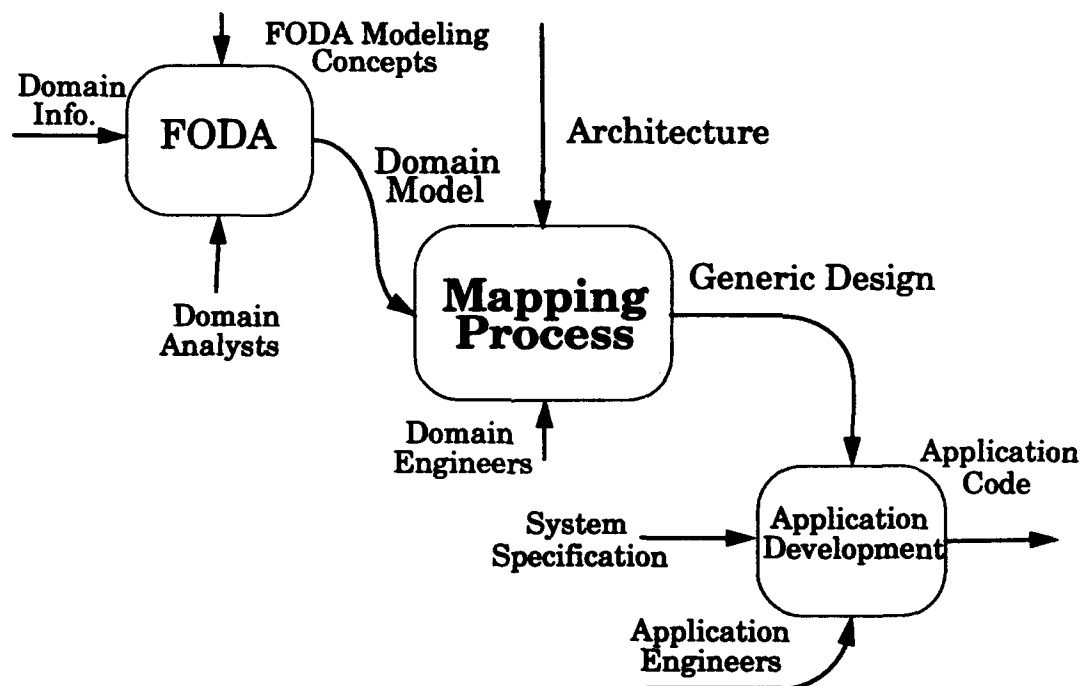


Figure 7-1: A Development Life Cycle Utilizing the Mapping Process

Figure 7-1 formalizes the roadmap depicted in Figure 1-1 at the beginning of this report and illustrates the use of the mapping process as an integral part of a development lifecycle where domain analysis provides the models needed to characterize the requirements for a related set of applications and the mapping process exploits the models to develop a generic design which is then reused for each product in the program family.

The authors completed, with the assistance of a graduate student who wrote the GUI, a demonstration prototype consisting of 22,000 lines of Ada code and 2,500 lines of "C". This prototype demonstrates the viability of the mapping process. Only a general notion of the desired capabilities were selected as features from the features model. The remaining data and detailed functional requirements were gathered following this process. The resulting

software contains four subsystems, four objects, two surrogates, two handlers, four transforms, and an executive. Of the 22,000 lines of Ada, approximately 6,500 lines were reuses of various Booch components, described in [Booch 87]. The example movement control software described throughout this report has been compiled and executed on multiple hardware/OS/compiler combinations.

7.2 Future Directions

7.2.1 Near-Term

1. Further validate the model(s) and templates by documenting their usage by outside users on their domain analysis or reusable software efforts. In particular, work with an organization in a domain with real-time and other performance related requirements to test the process's ability to incorporate and suitably handle such requirements.
2. For the example system, further prove the flexibility of the OCA by:
 - a. replacing the rudimentary map abstraction with calls to a more comprehensive Geographical Information System (GIS),
 - b. reimplementing the "C"-based GUI in Ada, using appropriate Ada bindings to X and Motif and removing the type conversion routines and substituting a more flexible buffer interface for the data items to be exchanged between the GUI and the rest of the application, and
 - c. reimplementing the I/O packages under the Data_Base surrogate to incorporate the use of a commercially available relational database using SQL syntax.
3. Extend the executive model from its present form with a single thread of control to incorporate the ability to handle multiple threads of control, thus providing the ability to handle multiple asynchronous devices.

7.2.2 Long-Term

1. Investigate mechanisms that will make possible the development of more highly reusable objects, including further studies involving the use of generics and the forthcoming changes/features of the new Ada standard, Ada9X.
2. Investigate the potential for automation of the process of template generation and completion through use of appropriate software tools. If this is successful, continue to explore the automation process towards the goals of complete generation of applications via selection of features and information entities desired.
3. Further investigate the processes described in this document through case studies. From analysis of such studies, refine the processes to strengthen their utility and understand their application in other domains.
4. Implement the OCA executive as a set of subsystems and surrogates that provide access to the following services:

- a. Event management, including time.
- b. Schedule management.
- c. Import/Export management, to allow for the dynamic binding of connections between the Import and Export packages.
- d. Control sequencing (subsystem activation), based upon schedule constraints and subsystem location.
- e. Registrar, responsible for the initialization, finalization, and location of subsystems and surrogates on an ongoing basis.

Such an implementation would make possible fully distributable versions of applications using the OCA, thus achieving the degree of flexibility needed in the design and implementation of software systems in the future.

The mapping process should be a useful addition to the development process of any organization looking to reap the benefits of domain analysis and the systematic exploitation of software architectures. The generic design and supporting components developed from use of a domain model and a selected architecture will greatly increase the reuse potential and maintainability of application instances within a domain due to the common parentage of their underlying software. Such increases translate into decreased long-term costs, an important part in creating a competitive advantage needed to survive in today's global software marketplace.

References

- [Abowd 93] Abowd, Gregory J.; Bass, Len; Howard, Larry; & Northrup, Linda. *Structural Modeling: An Application Framework and Development Process for Flight Simulators* (CMU/SEI-93-TR-14, ADA271348). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, August 1993.
- [Ada 83] Ada Joint Program Office, United States Department of Defense. *Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A). Washington, DC: GPO, 1983.
- [AHD 85] *The American Heritage Dictionary*, 2nd College Edition. William Morris, ed. Boston, MA: Houghton Mifflin Co., 1985.
- [Bardin 87a] Bardin, Bryce M. & Thompson, Christopher J. "Composable Ada Software Components and the Re-Export Paradigm." *ACM SIGAda Ada Letters*, Vol. 8, 1 (Jan. 1988): 58-79.
- [Bardin 87b] Bardin, Bryce M. & Thompson, Christopher J. "Using the Re-Export Paradigm to Build Composable Ada Software Components." *ACM SIGAda Ada Letters*, Vol. 8, 2 (March 1988): 39-54.
- [Booch 87] Booch, Grady. *Software Components with Ada: Structures, Tools, and Subsystems*. Menlo Park, CA: Benjamin Cummings, 1987.
- [Booch 93] Booch, Grady. *Object-Oriented Analysis and Design with Applications*. Menlo Park, CA: Benjamin Cummings, 1993.
- [Cohen 92] Cohen, Sholom G.; Stanley, Jay L., Jr.; Peterson, A. Spencer; & Krut, Robert W., Jr. *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain* (CMU/SEI-91-TR-28, ADA256590). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, June 1992.
- [Feiler 93] Feiler, Peter H. *Rengineering: An Engineering Problem* (CMU/SEI-93-SR-5). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, May 1993.
- [Fernandez 93] Fernandez, Jose L. *A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis* (CMU/SEI-93-TR-34). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, Dec. 1993.
- [Gautier 90] Gautier, Robert J. & Wallis, Peter J. L., eds. *Software Reuse with Ada*. London, England: Peter Peregrinus Ltd., 1990.
- [Gelemter 92] Gelemter, David & Carriero, Nicholas. "Coordination Languages and Their Significance." *Communications of the ACM*, Vol. 55, 2 (Feb. 1992): 97-107.

- [Hefley 92] Hefley, William E.; Foreman, John T.; Engle, Charles B. Jr.; & Goodenough, John. B. *Ada Adoption Handbook: A Program Manager's Guide*, 2nd ed. (CMU/SEI-92-TR-29, ADA258937). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, Oct. 1992.
- [Kang 90] Kang, Kyo C.; Cohen, Sholom G.; Hess, James A.; Novak, William E.; & Peterson, A. Spencer. *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU/SEI-90-TR-21, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, Nov. 1990.
- [Krut 93] Krut, Robert W., Jr. *Integrating OO1 Tool Support into the Feature-Oriented Domain Analysis Methodology* (CMU/SEI-93-TR-11). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, July 1993.
- [Lee 88] Lee, Kenneth J.; Rissman, Michael J.; D'Ippolito, Richard; Plinta, Charles; & Van Scoy, Roger. *An OOD Paradigm for Flight Simulators*, 2nd ed. (CMU/SEI-88-TR-30, ADA204849). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, Sept. 1988.
- [Marca 88] Marca, David A. & McGowan, Clement L. *SADT: Structured Analysis and Design Technique*. New York, NY: McGraw-Hill, 1988.
- [McNicholl 88] McNicholl, Dennis G.; Cohen, Sholom G.; Palmer, Constance; et. al. *Common Ada Missile Packages - Phase 2 (CAMP-2), Volume I: CAMP Parts and Parts Composition System*. (AFATL-TR-88-62). Eglin AFB, FL: Air Force Armament Laboratory, Nov. 1988. Note: Distribution limited to DoD and DoD contractors only.
- [OED 87] *The Oxford English Dictionary*, Compact Ed. R. W. Burchfield, ed. Vol. 3. Oxford, England: Clarendon Press, 1987.
- [Parnas 76] Parnas, David L. "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering*, Vol. TSE2, 1 (Jan. 1976): 1-9.
- [Prieto-Diaz 91] Prieto-Diaz, Rueben & Arango, Guillermo, eds. *Domain Analysis and Software Systems Modeling*. Los Alamitos, CA: IEEE Computer Society Press, 1991.
- [Shaw 90] Shaw, Mary. "Toward Higher-Level Abstraction for Software Systems." *Data and Knowledge Engineering* 5, p. 119-128. New York, NY: North Holland, 1990.
- [Simon 81] Simon, Herbert A. *The Sciences of the Artificial*, 2nd ed. Cambridge, MA: MIT Press, 1981.

- [Srinivas 91] Srinivas, Yellamraju V. "Algebraic Specifications for Domains." *Domain Analysis and Software Systems Modeling*, p. 90-124. Los Alamitos, CA: IEEE Computer Society Press, 1991.
- [Stroustrup 91] Stroustrup, Bjarne. *The C++ Programming Language*, 2nd ed. Reading, MA: Addison-Wesley, 1991.
- [USAF 93] United States Air Force, Aviation Systems Command. *An Introduction to Software Models* (USAF ASC-TR-93-5008). Dayton, OH: Wright Patterson AFB, 1993.
- [USArmy 90] United States Army. *Field Manual FM 55-10: Movement Control in a Theater of Operations*. Washington, DC: Headquarters, Department of the Army, Nov. 1990.
- [Wirth 85] Wirth, Nicklaus. *Programming in Modula-2*, 3rd, corrected ed. New York, NY: Springer-Verlag, 1985.
- [Withey 94] Withey, James V. *Implementing Model-Based Software Engineering in Your Organization: An Approach to Domain Engineering* (CMU/SEI-94-TR-1). Pittsburgh, PA: Software Engineering Institute, April 1994.
- [Zaremski 93] Zaremski, Amy M. & Wing, Jeanette M. *Signature Matching: A Key to Reuse* (CMU-CS-93-151). Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, May 1993.
- [001SRM] The 001™ Tool Suite. *System Reference Manual*, Ver. 3. Cambridge, MA: Hamilton Technologies, Inc., Jan. 1992.

Appendix A The Domain Design Process

This appendix contains the detailed descriptions for the process described in Chapter 4 of this report. The details involve the specifics of completing the applicable forms as found in Appendix D using the designated domain model information.

A.1 Select Features from Domain Model

No form data is completed at this point.

A.2 Create Object Specifications

A.2.1 Identify Objects

Using the *Object Specification Form* (shown in Appendix D.2), begin the documentation of the object entity by giving it an appropriate *Object Name* and a short *Description* of the entity to be refined in later steps.

A.2.2 Derive Object Operations and Input/Outputs

Document the operations derived from operational features directly into the *Features to be Supported* section of the Object Specification form and those operations needed to supported data flows in the Operational Model in the *Overview of Requirements* section of the same form. For the inputs and outputs for each operation as noted in the Operational Model, enter the information about them into the *Inputs* or *Outputs* section of the Object Specification form as appropriate. Finally, any special or error conditions that are needed to describe the status of an object before or after an operation is performed are noted in the *Exceptions/Malfunctions* section for appropriate consideration during further refinement and implementation.

A.3 Create Subsystem Specifications

Begin filling out a *Subsystem Specification Form* (shown in Appendix D.1) by entering the *Subsystem Name*. The information for the *Description* section can be taken directly from the textual information in the features catalog and the *Overview of Requirements* can be also be taken from the features catalog as well as specific wording from a requirements document. List the objects to be aggregated by the Subsystem under the *Objects* section by entering those objects which are subfeatures of the parent feature which is being allocated as a subsystem. For each object, copy in the required inputs and outputs from the Object Specification Form.

A.4 Create Surrogate Specifications for Logical or Physical Devices

Fill out a *Surrogate Specification Form* (shown in Appendix D.3). First, provide a name in the *Surrogate Name* field and a *Description*. Then, enter the *Type* information by selection of the control characteristics as described as above. The *Connection to I/O device* information is necessary to give the software developer the requirements to what kind of device the

surrogate is providing an abstraction for. For most devices, it will be important to indicate the *device name*, which can be a logical device like X11 or Motif for graphical displays or a database product like Ingres or Informix for data storage, or a physical devices, like a SCSI controller. For some devices, it will also be necessary to note the device's buffer capability in the *size of data buffer* area. The device-specific information may be deferred until later in the surrogate's development. Finally, the *Imports* and *Exports* information is entered, as appropriate, in terms of the *Name*, *Type*, and *Source* or *Destination* and the *Exception/Malfunction* information entered, in terms of *Name* and *Effect*.

Appendix B The Domain Implementation Process

This appendix contains the detailed descriptions for the process described in Chapter 5 of this report. The details involve the specifics of using the information on the specification forms to fill out code templates as found in Appendix E.

B.1 Identify or Create Applicable System Engineering Units Package(s)

No specific instructions are required at this point.

B.2 Create Subsystem *_Types* Package

The starting point is an Ada *_Types* package specification template, seen in Appendix E.2. The template provides a standard placeholder for the `<subsystem_name>` to be supplied by the user. The given `<subsystem_name>` will be used consistently throughout the succeeding steps that reference the subsystem-level implementation components. Create the necessary type definitions, using the type information given on the Subsystem Specification Form and its references to applicable parts of the information model.

B.3 Create Object Signatures Package

The Signatures package may not be needed at the object level if there are no appropriate features to be dealt with. If such features exist, in particular *alternative features*, use the *Object Manager Signatures* template as shown in Appendix E.5. Transfer the Object Name given on the *Object Specification Form* (completed as described previously in Sections 4.2.1 and 4.2.2) onto the template, replacing the `<object_name>` placeholder. For each group of independent features, map the features onto specific names in the enumeration list and give the enumeration type itself a name to replace the `<feature_group>` placeholder.

B.4 Create Object Manager Package Specification

The *Object Manager* code template (specification), shown in Appendix E.6, is the starting point for this step. First, fill in the context clause section by writing *with* statements for the packages that declare the types needed to complete the parameter profiles for the subprogram templates. Again, transfer the Object Name from the Specification Form onto the template, replacing the `<object_name>` placeholder.

For each class of error or malfunction listed in the Exceptions/Malfunions section of the Object Specification Form (or any otherwise erroneous conditions that potentially may propagate out of the subprograms and into a controller body), declare an Ada exception to be raised at applicable points within the object body code and to be handled by name with the controller body. Remember to document the possible users of the exception in the *Raised By* comment immediately after the exception declaration statement.

As a last check, review the *Overview of Requirements and Features to be Supported* sections of the Object Specification Form and verify that the given subprograms in the completed Object Manager specification support all of the required operational needs allocated to the objects.

B.5 Create Subsystem/Surrogate Signatures Package

B.5.1 Subsystem Signatures

Start with the *Subsystem Signatures* code template as shown in Appendix E.1. For each object that is to become part of the subsystem with a Signatures package, create the `with` statement to gain visibility to the package and its contents. As with the construction of the Object Signatures package, transfer the *Subsystem Name* from the Subsystem Specification Form onto the template, replacing the `<subsystem_name>` placeholder. The first major step in completing the Subsystem Signatures is to fill out the `Entity_Type` declaration by entering all of the applicable objects and the entities supported by them into the enumeration list. As discussed in Section 4.2.1, entities are all of those data items that must be manipulated by the application, regardless of whether or not they are allocated as actual objects in the resulting subsystems. It is the combination of these entity names and the operation names that will allow the subsystem to select the appropriate object operation.

The next step is the process of reexporting the information contained in the utilized Object Signatures packages, if any. This is a two part process:

1. Declare an Ada subtype using a feature enumeration type as the base type. For example,

```
subtype Route_Features is Router.Features_Type;
```

2. For each enumeration literal declared in the enumeration base type, declare a function that returns a value of the subtype which renames the enumeration literal. For example,

```
function Best return Router.Features_Types
renames Router.Best;
```

Lastly, incorporate the information found in the *Exceptions/Malfunions* section of the Subsystem Specification Form by completing the `Status_Type` declaration. For each item listed, create a corresponding enumeration literal in the enumeration type. Two of the predefined names in the enumeration list, `Initialized` and `Normal` (the first and last literals in the list), are important to the implementation templates and should not be removed or renamed. Note that the handling of the *Exceptions/Malfunions* items differs from how they were used in development of the Object Signatures and Handler specifications. The reasons for the differences in placement, implementation, and ultimate usage will be discussed in a subsequent report describing the OCA implementation in greater detail.

B.5.2 Surrogate Signatures

For surrogates, the process is similar, but with a couple of important modifications. The *Surrogate Signatures* code template is shown in Appendix E.11. The first important difference of note is that there are two distinct forms of surrogate, one for user interface (UI) devices and another for devices not associated with UIs. The information content varies greatly between the two forms. The nominal device surrogate Signatures package contains two enumeration types:

1. One lists the entities (data items) that are to be processed by the surrogate for input and output as appropriate to or from the functional subsystems.
2. The other lists the errors that the device is capable of generating.

Ultimate completion of the surrogate Signature packages (and the Controller specification and body) must be deferred until the application is defined. However, we can begin the process. In particular, the basic structure can be selected by removal of inappropriate template items and entry of the package name can be done by replacing the `<device_name>` placeholder with the *Surrogate Name* from the Surrogate Specification Form. Further work is deferred until the process described in Section 6.2.1 begins.

B.6 Create Subsystem/Surrogate Controller Package Specification

The *Subsystem Controller* code template (specification) is shown in Appendix E.3. This is an extremely simple template to complete, as there is only one unique placeholder, that for the `<subsystem_name>`, that must be replaced with the actual Subsystem name from the Specification Form, one for each occurrence of the placeholder in the template.

The *Surrogate Controller* code template (specification) is shown in Appendix E.12. Just as was done in the previous section we can begin the completion of the Controller Code template by replacing the `<device_name>` placeholders with the Name given on the Surrogate Specification Form.

B.7 Create Subsystem/Surrogate Import and Export Packages

The *Export* package code template is shown in Appendix E.8. The first step is to replace the occurrences of the `<subsystem_name>` placeholder with the appropriate Subsystem Name. Remove any unneeded with statements or add any additional references as determined from the *Type* information for the *Exports* section of the Specification Form. Then, for each data item listed in the Exports section, complete an instance of the template for declaring exported values, filling out the `<exported_value_name>` using the *Export Name* information on the form and the applicable data type and package information placeholders.

The *Import* package code template comes in two parts. The Specification portion is shown in Appendix E.9. To begin filling out this template, replace the `<device_name>` placeholder with the applicable Subsystem Name from the Specification Form. Also, fill in the names of the other subsystem or package placeholders where data types to be imported are declared. The use of the *Application_Signatures* package is needed only if there is a case where the same data item (by type name) can be imported from two or more sources, depending upon the current operation being performed at the executive level. Remove this reference if all imports come from unique sources, as determined by examination of the *Source* field of the *Imports* section of the Specification Form. Then, for each item listed in the *Imports Name* section, complete an instance of the import function template using the Name and Type information to obtain the needed replacement for placeholders.

When the Import package specification is completed, begin the work of filling out the corresponding Body template, shown in Appendix E.10. Here is where the important task of binding data exports to corresponding imports is performed. Begin by replacing the placeholders for the `<subsystem_name>` and `<other_export>`s using the *Subsystem Name* and the *Import Source* data to delineate the applicable export packages to be withed in (hence, the necessity to declare the export packages before the imports). Then, for each import function declared previously in the package specification, create an equivalent function body of the applicable type corresponding to the need to specify a *From* source. In most cases, the function body will simply return the data value exported by the data object in the Export package. In the case of multiple sources, the *From* parameter will be used to select the appropriate arm of the case statement and return the applicable data object from the corresponding Export package. Again, some of the *Source* export packages will not be available until the surrogates are completed later as described in Section 6.2.3 and Appendix C.2.3.

B.8 Create Subsystem/Surrogate Controller Package Body

The *Subsystem Controller* code template (body) is shown in Appendix E.4. Begin the completion of the template by replacing the `<subsystem_name>` placeholders that occur throughout the template. Note that when this replacement is performed, the Types, Imports, and Exports packages are now correctly named for use in the Controller package body code. Next, replace the `<object>` placeholders with appropriate references to Objects as defined in the Objects section of the Subsystem Specification Form. These two sets of replacements complete the context clause for the Controller body. The next step is to complete the bodies for the operational subprograms using the sequence of events information discussed in Section 5.8.

Events 1 and 2 are implemented by creation of an Ada case statement containing when clauses for each entity to be handled and using the *others* syntax with a *null* statement or error condition to complete the list of entity alternatives. Event 3 is implemented by filling in a call to the applicable Object operation for the entity to be handled and corresponding to the semantics of the subsystem call. Event 4 requires the use of an exception handler at some level within the procedure body. Depending on the effect desired, exception can be handled

within the block statement that encapsulate the Object call with a specific exception handler, or with a single exception handler just preceding the end of the subprogram being implemented.

Repeat this step as needed for each subprogram stub in the Controller body. Finally, add calls to applicable operations for any Objects that require explicit initialization in the final statement block prior to the initialization of the `Internal_State` variable.

The *Surrogate Controller* code template (body) is shown in Appendix E.13. After filling the placeholders and creating the body stubs for the procedures defined in the Controller specification, create the template by calling the operations needed to handle each call using the *Handlers* and *Transform* packages. The style of these package is dependent upon the implementation strategy used which is discussed in Section 5.6 of the report.

B.9 Create Object Manager Package Body

The *Object Manager* code template (body) is shown in Appendix E.7. After the replacement of the `<object_name>` placeholder and the `<subsystem_name>` placeholder to within the applicable subsystem *Types* package, the only other predefined step is to generate a body skeleton for each operation declared in the Object Manager specification (completed in Section 5.4).

Appendix C Using a Generic Design in Application Development

This appendix contains the detailed descriptions for the process described in Sections 6.1 - 6.3 of this report. The details involve the specifics of using the information on the specification forms to fill out code templates as found in Appendix E.

C.1 Create an Application Signatures Package

The *Application Signatures* code template is shown in Appendix E.14. First, the Application Aggregate enumeration type definition is completed by naming each subsystem and surrogate to be used in the application. This declaration will allow the executive to use the names of the subsystems and surrogates in its decision logic. Two subtype declarations provide the application with a single typename for use in describing the subsystem and surrogate subsets.

The next step is to define an enumeration type which defines the callable operations to be supplied by subsystems and their underlying objects. The template gives a predefined set of names: *Construct*, *Destruct*, and *Fetch*. These names correspond to those names given to the callable subprograms supplied in the subsystem controller and object manager templates.

The last step is to define an enumeration type to describe an appropriate namespace for the overall state of the application, i.e., the executive state. Again, the template predefines a useful three state system:

1. *Initialize* - the system state before the executive invokes any subsystems and surrogates to bring the system to a defined state of usability.
2. *Steady* - the system state in which the application is able to perform its intended function(s).
3. *Finalize* - the system state in which the application, if possible, shuts itself down in an orderly manner, saving system changes as listed before final exit.

Other states may be added as necessary for the executive to maintain an overall understanding of the state of the application.

C.2 Complete Packages Making Use of Application Signatures

C.2.1 Complete Surrogate Signatures Package

The first step in the process is to complete the *Imports* and *Exports* sections of the Surrogate Specification Form started in Section 4.4. Name all of the values exported by the subsystems for the surrogate's use (as listed in each subsystem's *Exports Destination* field) as *Imports* with the corresponding *Source*. Similarly, name each value to be imported by the subsystems whose *Source* is given as the applicable surrogate, and create a corresponding *Export* value

with the appropriate *Destination*. This information is needed to complete the Signatures package because the entities to be handled by the surrogate must be defined via the enumeration namespace.

C.2.2 Complete Surrogate Controller Package Specification

No additional description for implementation is needed at this point.

C.2.3 Complete Surrogate Import/Export Packages

See Appendix B.7 for details previously described.

C.2.4 Complete Subsystem Import Package Body

For each appropriate subsystem that can supply a needed data type for import, create a case arm and name the applicable data item *Name* from its Export package. Be sure to use the `when others => null;` clause to account for the unused subsystems.

C.2.5 Compete Surrogate Controller Package Body

The `Application_To_Device` and its converse `Device_To_Application` subprograms are filled in using a format equivalent to that given for the subsystem controller subprogram bodies, where the `ENTITY` parameter is used to select the appropriate branch/arm of a case statement to invoke the applicable subprograms from the Transforms and Handlers accessible from within the controller package body.

C.3 Complete the Executive Template

The Ada code template to use as a starting point is shown in Appendix E.15. Although the construction of the executive will involve the construction of a significant amount of code spanning many kinds of operations within an application, there is a recurring sequence of steps to follow for each executive operation:

1. If the operation is invoked due to a transfer of control (i.e., a Signal return from a surrogate), then use the appropriate `Device_To_Application` as needed to transform and move any associated data to the surrogate's Export package.
2. Determine if a control loop exists between two subsystems and/or surrogates with respect to the movement of multiple data items between. If one exists, determine the appropriate Signal return value from a subsystem to be used to terminate the loop.
3. Call the appropriate sequence of subsystem or surrogate subprograms to achieve the desired effect, using the operational model as the basis for determining what to call and in what order. The executive must pass the subsystem the appropriate Entity, and possible Feature, information.
4. Ensure any potential error conditions are checked for after subsystem operations by invoking the Signal function and comparing the results to the nominally expected Normal value.

5. If the operation's final result is to send output via a surrogate, use the appropriate `Application_To_Device` to initiate this output. Alternately, if a subsystem generates an error to be displayed by the UI device, the UI surrogate's `Application_To_Device` subprogram that uses the `Error_Return_Type` must be invoked.

Step 2 through 5 are applicable even in the initialization and finalization states of an application, as much processing by subsystems is performed to upload initial state from external storage, and, conversely, to download or verify final state prior to application termination.

It is possible to segment the levels of decomposition into smaller Ada program units, if desired, by using the "separate" facility of the language to create extra subprograms that still maintain full visibility of the executive's control information, most importantly, the `Status_Type` record.

Appendix D Specification Form Templates

D.1 Subsystem Specification Form

Subsystem Name: _____

Description: _____

Overview of Requirements:

Features to be Supported:

Objects:

_____	_____	_____
_____	_____	_____
_____	_____	_____

Imports:

Name

Type

Source

_____	_____	_____
_____	_____	_____
_____	_____	_____

Exports:

Name

Type

Destination

_____	_____	_____
_____	_____	_____
_____	_____	_____

Exceptions/Malfunctions:

Name

Effect

_____	_____
_____	_____
_____	_____

D.2 Object Specification Form

Object Name: _____

Description: _____

Overview of Requirements:

Features to be Supported:

Imports:

Name

Type

Exports:

Name

Type

Exceptions/Malfunctions:

Name

Effect

D.3 Surrogate Specification Form

Surrogate Name: _____

Description: _____

Type: **monitor** **control** (check one or both)

Connection to I/O device:

device name _____

size of data buffer (in bytes) _____

Imports (for monitor surrogate):

Name	Type	Source
_____	_____	_____
_____	_____	_____
_____	_____	_____

Exports (for control surrogate):

Name	Type	Destination
_____	_____	_____
_____	_____	_____
_____	_____	_____

Exceptions/Malfunctions:

Name	Effect
_____	_____
_____	_____
_____	_____

Appendix E Ada Code Templates

E.1 Subsystem Signatures Code Template

```
with <object_name>_Signatures; -- as needed for each object.
package <subsystem_name>_Signatures is

    -- every subsystem controller has to differentiate between
    -- the many objects and their parts that may be used.
    -- Objects may perform operations differently depending
    -- implemented or user-selected features, so entity names
    -- may be combinations of entity and feature identifiers.

    type Entity_Type is ( <Ent_Name_1>, <Ent_Name_2>,
                          <Ent_Name_3> .. <Ent_Name_n> );

    -- Subsystems may return errors and/or other Signal
    -- information. Always includes a "Normal" or "OK".
    -- This type can be extended to incorporate any error
    -- conditions to be propagated to the executive.

    type Status_Type is ( Initialized, Incomplete, Complete,
                          Invalid, ..... Normal );

    type Error_Type is record
        STATUS : Status_Type;
        ENTITY : Entity_Type;
    end record;

end <subsystem_name>_Signatures;
```

E.2 Subsystem _Types Code Template

```
package <subsystem_name>_Types is

    type <named_entity_type> is ....;

    -- Declare your types for import and export here if not
    -- declared elsewhere.

end <subsystem_name>_Types;
```

E.3 Subsystem Controller Code Template (Specification)

```
with Application_Signatures; -- if Source parameter used
with <subsystem_name>_Signatures;
package <subsystem_name>_Controller is

    -- Every subsystem controller has at least 3 procedures
    -- callable by the executive, derived from these below
    -- Optionally, the executive may require use of the
    -- SOURCE parameter if multiple sources exist for
    -- a particular data item.

    procedure Construct( ENTITY: in
                        <subsystem_name>_Signatures.Entity_Type
    --
                        SOURCE : in
    --
                        Application_Signatures.Subsystem_Type
                        );

    procedure Destruct( ENTITY: in
                       <subsystem_name>_Signatures.Entity_Type );

    procedure Fetch( ENTITY: in
                    <subsystem_name>_Signatures.Entity_Type );

    -- Additionally, each subsystem may provide means to
    -- provide control information to the executive

    function Signal return
        <subsystem_name>_Signatures.Status_Type;

end <subsystem_name>_Controller;
```

E.4 Subsystem Controller Code Template (Body)

```
with SKU; -- global types
with <subsystem_name>_Types; -- the 'local' types
with <subsystem_name>_Imports;
with <subsystem_name>_Exports;

-- all objects that are part of this subsystem
with <object1>_Manager;
with <objectn>_Manager;

package body <subsystem_name>_Controller is

    -- local variables declared here
    INTERNAL_STATE : <subsystem_name>_Signatures.Status_Type;

    procedure Construct( ENTITY: in
        <subsystem_name>_Signatures.Entity_Type;
        SOURCE : in Application_Signatures.Subsystem ) is
    begin
        case ENTITY is
            -- algorithm for choosing correct Entity Construct call
            end case;
        end Construct;

    procedure Destruct( Entity: in
        <subsystem_name>_Signatures.Entity_Type ) is
    begin
        -- algorithm for choosing correct Entity Destruct call
        end Destruct;

    procedure Fetch( Entity: in
        <subsystem_name>_Signatures.Entity_Type ) is
    begin
        -- algorithm for choosing correct Entity Fetch call, etc.
        end Fetch;

    function Signal return
        <subsystem_name>_Signatures.Status_Type is
        Status : <subsystem_name>_Signatures.Status_Type
            := INTERNAL_STATE;
    begin
        INTERNAL_STATE := <subsystem_name>_Signatures.NORMAL;
        return Status; -- return an appropriate value
    end Signal;

begin
    -- any initialization code goes before this statement
    INTERNAL_STATE := <subsystem_name>_Signatures.INITIALIZED;
end <subsystem_name>_Controller;
```

E.5 Object Manager Signatures Template

```
package <object_name>_Signatures is

    type <features_group> is ( <feature_1>, ..., <feature_n> );

end <object_name>_Signatures;
```

E.6 Object Manager Code Template (Specification)

```
with SEU; -- global types
with <object_name>_Signatures; -- if used!
with ...; -- other needed data types;
package <object_name>_Manager;

    -- The procedures below are overloaded as needed for each
    -- parameter profile. Add parameters to facilitate use of
    -- features in the Signatures package as required.

    procedure Construct (
        <In_Parameter_1>: in SEU.<In_Type_1>;
        <In_Parameter_2>: in SEU.<In_Type_2> );

    procedure Destruct (
        <In_Parameter_1>: in SEU.<In_Type_1>;
        <In_Parameter_2>: in SEU.<In_Type_2> );

    procedure Fetch (
        <In_Parameter_1>: in SEU.<In_Type_1>;
        <In_Parameter_2>: in SEU.<In_Type_2>;
        <Out_Parameter_1>: out SEU.<Out_Type_1>;
        <Out_Parameter_2>: out SEU.<Out_Type_2>);

    -- Export any error information as exceptions to
    -- calling Subsystem and name the subprogram(s)
    -- able to raise them.

    <Error_Condition_1> : exception;
    -- raised by ...

    <Error_Condition_n> : exception;

end <object_name>_Manager;
```

E.7 Object Manager Code Template (Body)

```
with SEU; -- global types
with <subsystem_name>_Types; -- 'local' types

package body <Object>_Manager is

    type <Local_Object> is ... ;

    -- declaration of state data
    <Object_Name> : <Local_Object>;

    procedure Construct (
        <In_Parameter_1>: in SEU.<In_Type_1>;
        <In_Parameter_2>: in SEU.<In_Type_2> ) is
    begin
        -- algorithm goes here
    end Construct;

    procedure Destruct (
        <In_Parameter_1>: in SEU.<In_Type_1>;
        <In_Parameter_2>: in SEU.<In_Type_2> ) is
    begin
        -- algorithm goes here
    end Destruct;

    procedure Fetch (
        <In_Parameter_1>: in SEU.<In_Type_1>;
        <In_Parameter_2>: in SEU.<In_Type_2>;
        <Out_Parameter_1>: out SEU.<Out_Type_1>
        <Out_Parameter_2>: out SEU.<Out_Type_2>) is
    begin
        -- algorithm goes here
    end Fetch;

end <Object>_Manager;
```

E.8 Export Package Code Template

```
with SEU; -- global types
with <subsystem_name>_Types; -- 'local' types

package <subsystem_name>_Exports is

    <exported_value_name_1> : SEU.<type_name>;

    <exported_value_name_n> :
        <subsystem_name>_Types.<type_name>;

end <subsystem_name>_Exports;
```

E.9 Import Package Code Template (Specification)

```
with SEU; -- global types
with Application_Signatures;
with <subsystem_1_name>_Types; -- other subsystem types
...
with <subsystem_N_name>_Types; -- as needed
package <subsystem_name>_Imports is

    function <import1> return <import1_type>;

    function <import2>
        ( From : in Application_Signatures.Subsystem )
        return <import2_type>;

end <subsystem_name>_Imports;
```

E.10 Import Package Code Template (Body)

```
with <other_export1>;
with <other_export2>;
package body <subsystem_name>_Imports is

    function <import1> return <import1_type> is
    begin
        return <other_export1_data>;
    end;

    function <import2>
        ( From : in Application_Signatures.Subsystem )
        return <import2_type> is
    begin
        case From is
            when <subsystem_X> =>
                return <other_export2_data>;
        end case;
    end;

end <subsystem_name>_Imports;
```

E.11 Surrogate Signatures Code Template

```
-- These 'withs' are needed only for UI device
with Application_Signatures;
with <subsystem_1>_Signatures;
...
with <subsystem_n>_Signatures;
package <device_name>_Signatures is

    -- Depending upon the number of items, the device
    -- can either declare its own Entities or use those names
    -- declared in other Signatures packages.

    -- Devices may return errors and/or other Signal
    -- information. Always includes a "Normal" or "OK".
    type Status_Value is (Initialized, ... , Normal);

    -- For the surrogate to a User Interface device, the
    -- Status_Value is a layer of records that provide the
    -- Executive with the user selected operations/options.

    type Status_Value( SUBSYSTEM :
        Application_Signatures.Subsystem_Type ) is
    record
        Operation :
            Application_Signatures.Operation_Type;
        case SUBSYSTEM is
            when <subsystem_1> =>
                <subsystem_1>_Entity :
                    <subsystem_1>_Entity_Type;
                ....
            when <subsystem_n> =>
                <subsystem_n>_Entity :
                    <subsystem_n>_Entity_Type;
            end case;
        end record;

    -- Also need to make the Error info. available to the UI

    type Error_Return_Type( SUBSYSTEM :
        Application_Signatures.Application_Aggregate ) is
    record
        case SUBSYSTEM is
            when <subsystem_1> =>
                <subsystem_1>_Error :
                    <subsystem_1>_Signatures.Error_Type;
                ....
            when <subsystem_n> =>
                <subsystem_n>_Error :
                    <subsystem_n>_Signatures.Error_Type;
            end case;
        end record;

    end <device_name>_Signatures;
```

E.12 Surrogate Controller Code Template (Specification)

```
with <device_name>_Signatures; -- Signal return values
with <subsystem_1>_Signatures; -- Subsystems to be handled
...
with <subsystem_n>_Signatures;
package <device_name>_Controller is

    -- Send application data (via Export) to device
    procedure Application_To_Device( Entity : in
        <subsystem_x>_Signatures.Entity_Type );

    -- Receive data from device
    procedure Device_To_Application( Entity : in
        <subsystem_x>_Signatures.Entity_Type );

    -- For the UI 'device', there is a notion of a Key which
    -- is sent in isolation so that the application can Search
    -- for a complex value based on the given Key.

    type Data_Kind is ( Key, Entity );

    procedure Device_To_Application(
        Entity : in <subsystem_x>_Signatures.
            Entity_Type;
        Kind : in Data_Kind );

    -- Receive status/error information from device
    function Signal return
        <device_name>_Signatures.Status_Value;

end <device_name>_Controller;
```


E.13 Surrogate Controller Code Template (Body)

```
package body <device_name>_Controller is

    procedure Application_To_Device( Entity : in
                                     <subsystem_x>_Signatures.Entity_Type ) is
    begin
        -- select proper transform algorithm
    end Application_To_Device;

    procedure Device_To_Application( Entity : in
                                     <subsystem_x>_Signatures.Entity_Type ) is
    begin
    end Device_To_Application;

    -- or

    procedure Device_To_Application(
        Entity : in <subsystem_x>_Signatures.Entity_Type;
        Kind : in Data_Kind ) is
    begin
    end Device_To_Application;

    -- Receive status/error information from device
    function Signal return
        <device_name>_Signatures.Status_Value is
    begin
        return ...;
    end Signal;

end <device_name>_Controller;
```

E.14 Application_Signatures Code Template

```
package Application_Signatures is

  type Application_Aggregate is (
    <surrogate_1>, .... <surrogate_n>,
    <subsystem_1>, <subsystem_2>,
    ..., <subsystem_n-1>, <subsystem_n> );

  subtype Surrogate_Type is Application_Aggregate
    range <surrogate_1> .. <surrogate_n>;

  subtype Subsystem_Type is Application_Aggregate
    range <subsystem_1> .. <subsystem_n>;

  -- The 'common' operation names!
  type Operation is ( Construct, Destruct, Fetch );

  type Application_State is ( Initialize, Steady, Finalize );

end Application_Signatures;
```

E.15 Executive Template

```
with <subsystem_1>_Signatures;
with <subsystem_n>_Signatures;
with Application_Signatures;

with <subsystem_1>_Controller;
with <subsystem_n>_Controller;

procedure Executive is

-- Renaming of all 'with'ed packages to improve readability

package APP renames Application_Signatures;

Program_State : APP.Application_State := APP.INITIALIZE;
User_Selection : xxs.Status_Type;

begin
-- Call any needed initialization procedures here!

Program_State := APP.STEADY;

while Program_State = APP.STEADY loop
    User_Selection := UIC.Signal;
    case User_Selection.Subsystem is
        when APP.<subsystem_1> =>
            case User_Selection.Operation is
                when APP.Construct =>
                    case User_Selection.<entity_name> is
                        when xxs.<entity_1> =>

                            case User_Selection.<feature_group> is
                                when xxs.<feature_name> =>

                                    end case;

                                    when xxs.<entity_n> =>
                                        xxC.Construct( <entity_n> );

                                    end case;

                                when APP.Destruct =>

                                when APP.Fetch =>

                                end case;
                            end case;
                        end loop;

-- Call any finalization procedure here;

end Executive;
```


Appendix F Implementation Issues Affecting Reuse

Appendix F discusses some of the implementation issues dealt with during the trial usage of these processes, focusing on Ada language interface issues, and the idiosyncrasies found in implementations of Ada input/output packages. [Hefley 92] contains more information about other such issues involved in the use of Ada. This appendix also provides some specific examples of "C"¹ code used in the user interface portion of the movement control prototype used as the example case in the report, focused mainly on the description of several reusable abstractions for X/Motif input and output.

F.1 Interfaces to Other Languages/Environments

One of the perceived strengths of Ada is its ability to interface to code written in other programming languages. However, this strength is not without caveats. First, the Ada language's interface capability is NOT required to be supported by valid Ada compilers.² Second, even if the compiler does support the interface capability, how the interface is implemented is vendor dependent, other than the required use of the `pragma Interface`. Third, the reverse capability of calling Ada from other languages is not defined with the language.³ None the less, most Ada implementations are providing the interface capability and many of the vendors are using a standard (yet still somewhat ad hoc) nomenclature for their interface pragmas.

F.1.1 Use of X11 and Motif with Ada

Access to Ada bindings for X11 and Motif is the preferred means for utilizing the powerful functionality provided by these pieces of software for creating a Graphical User Interface (GUI) to be used in highly interactive applications. Unfortunately, not everyone has access (due to cost considerations, chief among many reasons) to usable bindings for current, i.e., widely used, versions of X11 or Motif.⁴ However, due to the interface capabilities described above, one can write a GUI using X and Motif calls in the C language. There are two major issues involved in doing this:

1. The X/Motif event loop must be able to respond to user events (i.e., mouse movements, button or key presses and releases) in a thread of control separate from that maintained by the executive and other subsystems. This need requires that Ada's tasking mechanisms be used to provide the ability to handle multiple threads of control.

1. The "C" programming language will be referred to hereafter without the use of quotation marks for brevity.

2. See [Ada 83], 13.9(4).

3. See [Ada 83], 13.9(6).

4. X11R5 and Motif v1.1 at the time of this report.

2. It is difficult, if not impossible, to ensure that various C implementations use the same internal representation for `structs`, the Ada equivalent of records. The equivalent of the Ada representation specification clause, documented in Section 13.4 of [Ada 83], is not available in C. Even though most C compilers do not attempt to reorganize `structs` in order to optimize the storage size or alignment of internal fields, there is no requirement that they maintain the representation given. Thus, there is no guarantee that data structures are portable across multiple platforms and compilers.

The net result is that there is no way to ensure that data structures written to be transferable across languages in one environment will be reusable in another environment. Therefore, the lowest common denominator solution is to choose a single data type that can hold, in theory, information of any other data type and have the language transform important data structures into and out of the single data type.

The common data type is the *string*, an array of or pointer to a sequence of characters, which in C is logically terminated by the ASCII NUL value (zero) and in Ada by the fixed size of the array. Both languages provide useful functions that take numeric-based data and transform it into string equivalents and vice versa. As long as the code in both languages knows the order in which the data is embedded in the string, each can maintain record structures for internal use, but also transfer data between each other using the string as the common structure.

Section F.3 documents some C/X/Motif subprograms that are reused, in some cases dozens of times, throughout the GUI code in the movement control example.⁵ The Print and conversion functions described therein provide some of the mechanisms used to deal with issues related to the use of strings as the data transfer method.

F.1.2 Calling C Within Ada and Ada Within C

To call Ada code from C, the C language requires that the subprogram be specified using the `extern` notation to identify the subprogram whose implementation must be matched with the specification at link time. Similarly, the Ada language uses compiler directives called pragmas to identify subprograms whose implementation (body) will be either supplied by another language (importation of functionality), or whose implementation satisfies the needs of a subprogram specified in another language (exportation of functionality).

F.1.3 Ada Tasking in an Application with a C/"X"-Based GUI

The end result of using all of the capabilities listed in the previous sections is a tasking architecture for systems with a GUI illustrated in Figure F-1.

This figure shows a design utilizing four Ada tasks. Two of the tasks, Blocking Input and Blocking Output, are passive buffer tasks that provide a mailbox capability for data, i.e., strings. They are passive in that they do not make any calls, they are only called by their users,

⁵ Due to the size (2000+ lines), the complete code for the GUI is not included in Appendix H where example code is shown.

as shown by the small control arrows in the figure. The Blocking feature is used to ensure that if one data block is sent, another data block cannot be sent until the first has been received, i.e., removed from the mailbox. This ensures that no data is lost and that data blocks are received and used in the correct order. The other two tasks are active, the Executive (the main program task) and the GUI. This Ada task utilizes the Ada calling C capability to run the C main procedure, which maintains the X Event loop as required. The C calling Ada capability is used in the implementation of the *sendbuf* and *rcvbuf* routines, which really are calls to Ada task entries for the Blocking Input and Output tasks.

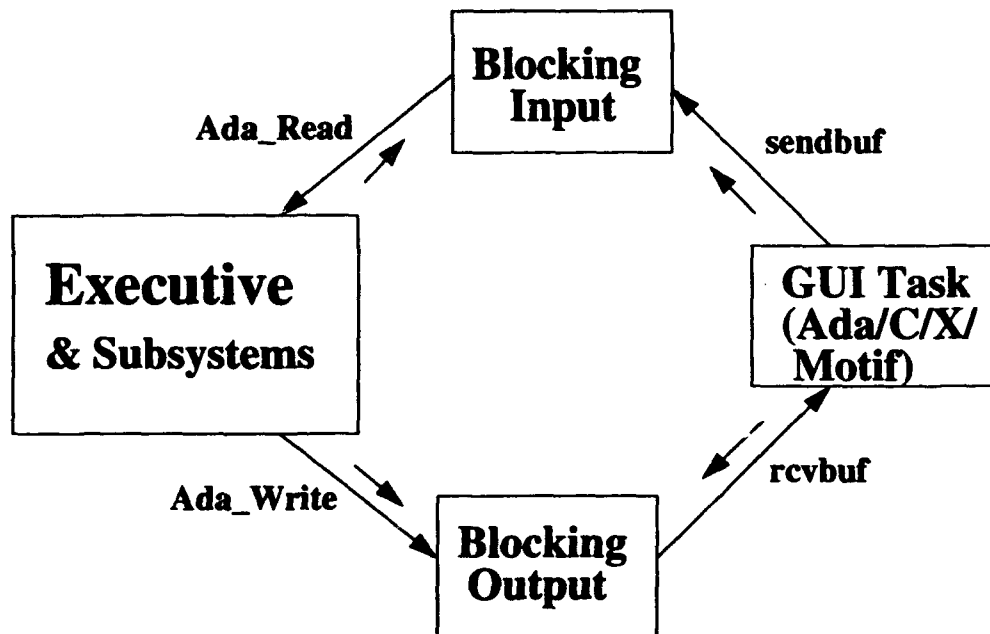


Figure F-1: Tasking Architecture Using a Separate X Event Loop

F.2 Using Ada I/O for Files

Another area where many current Ada compilers present some problems in the implementation of the example code is in the use of Ada file formats. Some mechanism was needed for maintaining some persistent storage of useful data, like maps, organized convoys, vehicle data, etc. In full implementations of movement control systems, a database system, usually a relational database, is the preferred means for storing and accessing large amounts of interrelated data. Because of the nature of the project and the complexity of designing a suitable database schema, the example code uses file handling capabilities provided by the Ada language. In particular, a package called *Direct_IO* provides (in theory) random access to data in file via use of a *COUNT* value that produces a unique key to provide more direct access to records. Unfortunately, at least one compiler had problems in its *Direct_IO* implementation when Ada variant records of different sizes were required in the same files.

One implementation was unable to keep track of a suitable End_Of_File (EOF) marker, and thus, raised various exceptions when the EOF was not found. Therefore, alternate implementations of some of the subsystem I/O packages were required, using the Sequential_IO package as the basis for storing data and using additional Text_io files to store some information about the number of records, etc. that Direct_IO provides intrinsically. Even then, some compilers require use of the implementation-dependent file attribute, FORM, to make the file format needed acceptable.

F.3 Motif/X/C Code Examples

This section presents some examples of code developed for the initial version of the GUI for the movement control prototype built to illustrate the utility of the Mapping Processes. The code and these descriptions were written by Mr. Greg Walker, a summer intern on this project in 1992.

F.3.1 Print Function

The function `mySprintf()` works a lot like the standard C I/O function `sprintf()` except that the result is written in a static buffer. This relieves the calling routines of having to declare a buffer. The drawbacks are that subsequent calls to `mySprintf()` overwrite the previous call's results and the buffer has a fixed size. `mySprintf()` enables code fragments such as:

```
char buffer[64];
sprintf(buffer, "the answer is %d", 56);
doSomethingWith(buffer);
```

to be rewritten as:

```
doSomethingWith(mySprintf("the answer is %d", 56));
```

F.3.2 Converting Strings Between X11 and C

The function `unXmString()` is a wrapper around `XmStringGetLtoR()` which converts XmStrings into C strings. Each call to `unXmString()` frees the result of the previous call. Like `mySprintf()`, subsequent calls to `unXmString()` destroy the results of previous calls. `unXmString` enables code fragments such as:

```
char *text;
XmStringGetLtoR(string, XmSTRING_DEFAULT_CHARSET, &text);
doSomethingWith(text);
XtFree(text);
```

to be rewritten as:

```
doSomethingWith(unXmString(string));
```


The function `xmString()` is a wrapper around `XmStringCreateLtoR()` which converts C strings into XmStrings and it also relieves the caller from having to free the result. It gets around the problem of subsequent calls destroying the results of previous calls by keeping, in an array, the last 100 results generated. The 101st call to `xmString` frees the 1st result, the 102nd call frees the 2nd result, and so on. This way fragments such as:

```
doSomethingWith(xmString("Hello"),xmString("World"))
```

will work in a desirable manner, that is, the 2nd call to `xmString()` does not destroy the result of the 1st call.

F.3.3 Routines Utilizing Abstractions of Motif Widgets

The routine `feedListWidget()` is a wrapper around `XmListAddItemUnselected()` which accepts an array of items to be added. Also, it accepts the items as C strings and takes care of converting them to XmStrings. This is used primarily in the `input()` described below.

The routine `popupMessage()` creates a message dialog shell/box, displays a message, and returns the users response. The parent argument is a widget to be used as the parent for the dialog; the dialog will probably pop up on top of it. The type specifies the symbol to be displayed beside the message; it must be one of the valid `dialogTypes` for a Motif `MessageBox`. If the type is `XmDIALOG_ERROR`, `XmDIALOG_INFORMATION`, or `XmDIALOG_MESSAGE`, the cancel button will not be displayed. The message argument is a C string.

`popupMessage()` returns 0 if the user clicks Ok or -1 if the user clicks Cancel. A problem with this routine is that in order to prevent the user from doing anything else with the application until he has answered the dialog, the root widget's sensitivity is set to False; this causes some widgets to change their appearance.

The routines `popupCancelCallBack()` and `popupErrorMessage()` are trivial helper functions. `popupErrorMessage()` is a wrapper which sets the type to `XmDIALOG_ERROR`.

The `input()` function is a generalized means for creating dialog boxes which allow the user to enter/edit information. The dialog boxes can contain fields of various types which are constructed using enumeration literals and parameters as follows:

```
input(parentWidget,
      iLabel,"Hello world.",
      iBoolean,"this is a boolean",&booleanVariable,
      iEnum,"zero","one","two","three",NULL,&intVariable,
      iString,"Name:",&charPointerVariable,
      iInt,"Age:",&intVariable,
      iFloat,"Your best estimate of pi",&floatVariable,
      iNewColumn,
      iSelection,arrayOfStrings,sizeofArray,&charPtrVar,
      NULL);
```

Each of the field types can be used as follows:

- `iLabel` creates a label widget displaying the given string.
- `iBoolean` creates a togglebutton displaying the given string. The button's state is initially set to the value of the given boolean variable. If and when the user clicks Ok, the boolean variable is updated to the current value of the togglebutton; the variable is left unchanged if the user clicks Cancel.
- `iEnum` creates a radioBox using the given NULL terminated list of strings. The value of the `intVariable` determines which of the radioButtons is initially set. If and when the user clicks Ok, the `intVariable` is updated to reflect that radioButton which is currently set; if the user clicks Cancel, the `intVariable` is left unchanged.
- `iString`, `iInt`, and `iFloat` will each create a textfield with a label beside it, allowing the user to edit a string/int/float. In the cases of `iInt` and `iFloat`, the variable will not be changed if the user types in something that is not a number. In the case of `iString`, the initial value of the variable is assumed to be a malloc'ed C string. If the user changes it and clicks Ok, the old value will be free'd and space for the new value will be malloc'ed.
- `iNewColumn` says that a new column should be created and subsequent fields should be placed in it. All the previous fields were arranged in a single vertical column.
- `iSelection` creates a scrolled list using the given `arrayOfStrings`, and also creates a textfield which is managed a lot like a textfield created by `iString` except that the user can also set the string by clicking on one of the items in the list.
- The list of field declarations must be NULL terminated.

`input()` accepts a variable number of arguments so that arbitrarily large dialog boxes may be created. `input()` makes no attempt to check its arguments for errors; the programming utilities `cc` and `lint` won't help either.

`input()` assumes that the `*XmList.visibleItemCount` resource is set to a reasonable value. This is done in the `fallbackResources`. `input()` issues the same return values, and has the same problem with sensitivity as `popupMessage()`.

The routine `drawIcon()` draws a pixmap (icon) whose dimensions are `width*height`, centered about the point `(x,y)`. It also centers the label underneath the pixmap.

Appendix G Sample Completed Specification Forms

G.1 Asset Manager Subsystem Specification

Subsystem Name:

Asset_Manager

Description:

Manages the assets involved in movement planning (vehicles, transportation networks, equipment).

Overview of Requirements:

Manage information about military vehicles and combinations of vehicles.

Source: CMU/SEI-91-TR-28, section D.2.4.5.

Features to be Supported:

Determination of vehicles needed to facilitate a move or series of moves, in support of the planning of movement operations.

Source: CMU/SEI-91-TR-28, section E.1.1.3.

Objects:

Vehicle

Imports:

Name	Type	Source
Model_Or_Combo_ID	Vehicle_Types.Model_Type	User_Interface
Vehicle_ID	Vehicle_Types.Specific_Vehicle_Id	User_Interface
Vehicle	Vehicle_Types.Vehicle_Type	User_Interface, Data_Base
Combination	Vehicle_Types.Vehicle_Combination	Data_Base

Exports:

Name	Type	Destination
Vehicle	Vehicle_Types.Vehicle_Type	User_Interface, Data_Base
Vehicle_Combination	Vehicle_Types.Vehicle_Combination	Convoy_Builder, Data_Base

Exceptions/Malfunctions:

Name	Effect
Asset_Manager_Signatures.INVALID	Fetch of specific asset information not found in object

G.1.1 Vehicle Object Specification

Object Name:

Vehicle

Description:

Stores information about vehicles and combinations of vehicles.

Overview of Requirements:

Information includes vehicle ID, type, width, height, length, and load-carrying capacity.

Source: CMU/SEI-91-TR-28, section D.2.4.5.2.3.

Features to be Supported:

Allow the user to enter, delete, and find vehicle type and other composition data (e.g. height, width, weight) relevant to convoy building.

Source: CMU/SEI-91-TR-28, sections E.1.1.3.1.1 and E.1.1.4.1.1.4.

Imports:

Name	Type
Model_Or_Combo_ID	Vehicle_Types.Model_Type
Vehicle_ID	Vehicle_Types.Specific_Vehicle_Id
Vehicle	Vehicle_Types.Vehicle_Type
Combination	Vehicle_Types.Vehicle_Combination

Exports:

Name	Type
Vehicle	Vehicle_Types.Vehicle_Type
Vehicle_Combination	Vehicle_Types.Vehicle_Combination

Exceptions/Malfunions:

Name	Effect
Not_Found	Fetch of information relating to vehicles or combinations abandoned because of missing or incorrect data.

G.2 Data Base Surrogate Specification

Surrogate Name:

Data_Base

Description:

Provides an interface between a physical external data base and the other subsystems comprising the Convoy Planner prototype. This surrogate manages the creation, deletion, reading, and updating of database items. The prototype implementation uses stored files, rather than an actual database management system.

Type (check one or both)		Connection to I/O Device	
Monitor	Control	Device Name	Data Buffer Size (bytes)
Yes	Yes	Ada file I/O	N/A

Imports: (for control surrogates)

Name	Type
Vehicle	Vehicle_Types.Vehicle_Type
Combination	Vehicle_Types.Vehicle_Combination
Map_Name	String
Vertex	Map_Types.Vertex_Type
Arc	Map_Types.Arc_Type
Logical_Id_Value	Natural
Convoy_Name	Data_Base_Types.Convoy_Element_Type
Convoy_Parameters	Data_Base_Types.Convoy_Parameters_Type

Exports: (for monitor surrogates)

Name	Type
Records_In_File	Natural
Model_Id	Vehicle_Types.Model_Type
Vehicle_Id	Vehicle_Types.Specific_Vehicle_Id
Vehicle	Vehicle_Types.Vehicle_Type

Name	Type
Combination	Vehicle_Types.Vehicle_Combination
Convoy_Name	String(1 .. DB_Types.Max_Name_Length)
Convoy_Part	DB_Types.Convoy_Element_Type
Convoy_Data	DB_Types.Convoy_Parameters_Type
Map_Name	String(1 .. DB_Types.Max_Name_Length)
Vertex_Data	Map_Types.Vertex_Type
Arc_Data	Map_Types.Arc_Type
Logical_Id_Value	Natural

Exceptions/Malfunctions:

Name	Effect
Data_Base_Signatures.NOT_FOUND	Opening of specified file abandoned.

Appendix H Movement Control Example Code

This appendix contains a large sample of code to illustrate the application of the processes described in this document using a FODA domain model and the OCA as the architecture. The executive, one subsystem (the Asset Manager) with its objects and supporting components, and one surrogate (the Data_Base) with its Handler/IO packages are listed in the following sections.

H.1 Executive

```
with System;
```

```
with Application_Signatures;  
with User_Interface_Signatures;  
with Data_Base_Signatures;  
with Mapper_Signatures;  
with Asset_Manager_Signatures;  
with Convoy_Signatures;  
with Convoy_Builder_Signatures;  
with March_Table_Signatures;
```

```
with User_Interface_Controller;  
with Data_Base_Controller;  
with Mapper_Controller;  
with Asset_Manager_Controller;  
with Convoy_Builder_Controller;  
with March_Table_Controller;
```

```
procedure Executive is
```

```
-- Renamings of important packages
```

```
package APP renames Application_Signatures;  
package UIS renames User_Interface_Signatures;  
package DBS renames Data_Base_Signatures;  
package MPS renames Mapper_Signatures;  
package AMS renames Asset_Manager_Signatures;  
package CS renames Convoy_Signatures;  
package CBS renames Convoy_Builder_Signatures;  
package MTS renames March_Table_Signatures;
```

```
package UIC renames User_Interface_Controller;  
package DBC renames Data_Base_Controller;  
package MPC renames Mapper_Controller;  
package AMC renames Asset_Manager_Controller;  
package CBC renames Convoy_Builder_Controller;  
package MTC renames March_Table_Controller;
```

```
function "=" ( L, R : in APP.Status_Type ) return Boolean renames APP."=";  
function "=" ( L, R : in AMS.Status_Type ) return Boolean renames AMS."=";  
function "=" ( L, R : in DBS.Status_Type ) return Boolean renames DBS."=";  
function "=" ( L, R : in CBS.Status_Type ) return Boolean renames CBS."=";  
function "=" ( L, R : in MPS.Status_Type ) return Boolean renames MPS."=";  
function "=" ( L, R : in MTS.Status_Type ) return Boolean renames MTS."=";
```

```
Program_State : APP.Status_Type := APP.INITIALIZE;
```

```

    User_Selection : UIS.Status_Type;

begin -- Executive

-- Initialization code to read Vehicle files and the Convoy and Map lists.
DBC.Application_To_Device( ENTITY => DBS.MODEL_LIST,
                           STATUS => Program_State );
UIC.Application_To_Device( DBS.MODEL_ID );
loop
    DBC.Device_To_Application( DBS.MODEL );
    exit when DBC.Signal = DBS.END_OF_FILE;
    DBC.Device_To_Application( DBS.MODEL_ID );
    AMC.Construct( ENTITY => AMS.MODEL,
                  SURROGATE => APP.DATA_BASE );
    UIC.Application_To_Device( DBS.MODEL_ID );
end loop;

DBC.Application_To_Device( ENTITY => DBS.VEHICLE_LIST,
                           STATUS => Program_State );
UIC.Application_To_Device( DBS.VEHICLE_ID );
loop
    DBC.Device_To_Application( DBS.SPECIFIC_VEHICLE );
    exit when DBC.Signal = DBS.END_OF_FILE;
    DBC.Device_To_Application( DBS.VEHICLE_ID );
    -- AMC.Construct( ENTITY => AMS.SPECIFIC_VEHICLE,
    --                SURROGATE => APP.DATA_BASE );
    UIC.Application_To_Device( DBS.VEHICLE_ID );
end loop;

DBC.Application_To_Device( ENTITY => DBS.COMBINATION_LIST,
                           STATUS => Program_State );
UIC.Application_To_Device( DBS.COMBINATION_ID );
loop
    DBC.Device_To_Application( DBS.VEHICLE_COMBINATION );
    exit when DBC.Signal = DBS.END_OF_FILE;
    DBC.Device_To_Application( DBS.COMBINATION_ID );
    -- AMC.Construct( ENTITY => AMS.VEHICLE_COMBINATION,
    --                SURROGATE => APP.DATA_BASE );
    UIC.Application_To_Device( DBS.COMBINATION_ID );
end loop;

DBC.Application_To_Device( ENTITY => DBS.MAP_LIST,
                           STATUS => Program_State );
UIC.Application_To_Device( DBS.MAP_NAME );
loop
    DBC.Device_To_Application( DBS.MAP_NAME );
    exit when DBC.Signal = DBS.END_OF_FILE;
    UIC.Application_To_Device( DBS.MAP_NAME );
end loop;

DBC.Application_To_Device( ENTITY => DBS.CONVOY_LIST,
                           STATUS => Program_State );
UIC.Application_To_Device( DBS.CONVOY_NAME );
loop
    DBC.Device_To_Application( DBS.CONVOY_NAME );
    exit when DBC.Signal = DBS.END_OF_FILE;
    UIC.Application_To_Device( DBS.CONVOY_NAME );
end loop;

```

```

-- Initialization complete, give the user CONTROL!
Program_State := APP.STEADY;

while Program_State = APP.STEADY loop

    User_Selection := UIC.Signal;
    case User_Selection.SUBSYSTEM is
        when APP.USER_INTERFACE =>
            case User_Selection.OPERATION is
                when APP.DESTRUCT =>
                    Program_State := APP.FINALIZE; -- This is QUIT!!!!!!

                when others => null;
            end case;

        when APP.DATA_BASE =>
            case User_Selection.OPERATION is
                when APP.CONSTRUCT => null;

                when APP.DESTRUCT =>
                    case User_Selection.DATA_BASE_ENTITY is
                        when DBS.MAP =>
                            UIC.Device_To_Application( DBS.MAP_NAME );
                            DBC.Application_To_Device( DBS.MAP_NAME );

                        when DBS.CONVOY =>
                            UIC.Device_To_Application( DBS.CONVOY_NAME );
                            DBC.Application_To_Device( DBS.CONVOY_NAME );

                        when others => null;
                    end case;

                when APP.FETCH =>
                    case User_Selection.DATA_BASE_ENTITY is
                        when DBS.MAP =>
                            MPC.Destruct( MPS.MAP );
                            UIC.Device_To_Application( DBS.MAP_NAME );
                            DBC.Application_To_Device( ENTITY => DBS.MAP,
                                                    STATUS => APP.INITIALIZE );

                        if DBC.SIGNAL = DBS.NORMAL then
                            DBC.Device_To_Application( DBS.VERTEX );
                            UIC.Application_To_Device( DBS.VERTEX );
                            loop
                                DBC.Device_To_Application( DBS.VERTEX );
                                exit when DBC.Signal = DBS.END_OF_FILE;
                                UIC.Application_To_Device( DBS.VERTEX );
                                MPC.Construct( ENTITY => MPS.VERTEX,
                                                SOURCE => APP.DATA_BASE );
                            end loop;

                            DBC.Device_To_Application( DBS.ARC );
                            UIC.Application_To_Device( DBS.ARC );
                            loop
                                DBC.Device_To_Application( DBS.ARC );
                                exit when DBC.Signal = DBS.END_OF_FILE;
                                UIC.Application_To_Device( DBS.ARC );
                                MPC.Construct( ENTITY => MPS.ARC,
                                                SOURCE => APP.DATA_BASE );
                            end loop;
                    end case;
            end case;
    end case;
end while;

```

```

end loop;

DBC.Device_To_Application( DBS.LOGICAL_ID ;
UIC.Application_To_Device( DBS.LOGICAL_ID );
DBC.Application_To_Device( ENTITY => DBS.MAP,
STATUS => APP.FINALIZE );

else
UIC.Application_To_Device( ( SUBSYSTEM =>
APP.DATA_BASE,
DATA_BASE_ERROR =>
( ENTITY => DBS.MAP,
STATUS => DBS.NOT_FOUND ) ) );

end if;

when DBS.CONVOY =>
CBC.Destruct( CBS.Convoy );
UIC.Device_To_Application( DBS.CONVOY_NAME );
DBC.Application_To_Device( ENTITY => DBS.CONVOY,
STATUS => APP.INITIALIZE );
DBC.Device_To_Application( DBS.ELEMENT );
UIC.Application_To_Device( DBS.ELEMENT );
loop
DBC.Device_To_Application( DBS.ELEMENT );
exit when DBC.Signal = DBS.END_OF_FILE;
UIC.Application_To_Device( DBS.ELEMENT );
CBC.Construct( CBS.CONVOY_PART );
end loop;
DBC.Device_To_Application( DBS.PARAMETERS );
CBC.Construct( CBS.CONVOY_PARAMETERS );
DBC.Application_To_Device( ENTITY => DBS.CONVOY,
STATUS => APP.FINALIZE );

when others => null;
end case;
end case;

when APP.CONVOY_BUILDER =>
case User_Selection.OPERATION is
when APP.CONSTRUCT =>
case User_Selection.CONVOY_BUILDER_ENTITY.ENTITY is
when CBS.SUBUNIT =>
UIC.Device_To_Application( CBS.SUBUNIT );
CBC.Construct( CBS.SUBUNIT );

when CBS.VEHICLE =>
UIC.Device_To_Application( CBS.VEHICLE );
case User_Selection.CONVOY_BUILDER_ENTITY.IMPORT
is
when AMS.MODEL =>
AMC.Fetch( ENTITY => AMS.MODEL,
AS_TYPE =>
AMS.VEHICLE_COMBINATION );

when AMS.SPECIFIC_VEHICLE =>
AMC.Fetch( ENTITY => AMS.SPECIFIC_VEHICLE,
AS_TYPE =>
AMS.VEHICLE_COMBINATION );

when AMS.VEHICLE_COMBINATION =>

```

```

        AMC.Fetch( AMS.VEHICLE_COMBINATION );

        when others => null;
    end case;
    CBC.Construct( CBS.VEHICLE );

when CBS.SPEED =>
    UIC.Device_To_Application( CBS.SPEED );
    CBC.Construct( CBS.SPEED );

when CBS.GAP_DISTANCE =>
    UIC.Device_To_Application( CBS.GAP_DISTANCE );
    CBC.Construct( CBS.GAP_DISTANCE );

when CBS.GAP_DISTANCES =>
    UIC.Device_To_Application( CBS.GAP_DISTANCES );
    CBC.Construct( CBS.GAP_DISTANCES );

when CBS.GAP_MULTIPLIER =>
    UIC.Device_To_Application( CBS.GAP_MULTIPLIER );
    CBC.Construct( CBS.GAP_MULTIPLIER );

when CBS.GAP_MULTIPLIERS =>
    UIC.Device_To_Application( CBS.GAP_MULTIPLIERS );
    CBC.Construct( CBS.GAP_MULTIPLIERS );

when CBS.CONVOY =>
    case User_Selection.CONVOY_BUILDER_ENTITY .FEATURE
is
        when CS.FIXED =>
            CBC.Construct( CBS.FIXED );
        when CS.GOVERNED =>
            CBC.Construct( CBS.GOVERNED );
        end case;

    when others => null;
end case;

when APP.DESTRUCT =>
    case User_Selection.CONVOY_BUILDER_ENTITY.ENTITY is
    when CBS.SUBUNIT =>
        UIC.Device_To_Application( CBS.SUBUNIT );
        CBC.Destruct( CBS.SUBUNIT );

    when CBS.VEHICLE =>
        UIC.Device_To_Application( CBS.VEHICLE );
        CBC.Destruct( CBS.VEHICLE );

    when CBS.CONVOY =>
        CBC.Destruct( CBS.CONVOY );
        DBC.Application_To_Device( ENTITY => DBS.CONVOY,
                                STATUS => APP.FINALIZE );

    when others => null;
    end case;

when APP.FETCH =>
    case User_Selection.CONVOY_BUILDER_ENTITY.ENTITY is
    when CBS.GAP_KIND =>

```

```

        CBC.Fetch( CBS.GAP_KIND );
        UIC.Application_To_Device( CBS.GAP_KIND );

    when CBS.SPEED =>
        CBC.Fetch( CBS.SPEED );
        UIC.Application_To_Device( CBS.SPEED );

    when CBS.GAP_DISTANCE =>
        CBC.Fetch( CBS.GAP_DISTANCE );
        UIC.Application_To_Device( CBS.GAP_DISTANCE );

    when CBS.GAP_DISTANCES =>
        CBC.Fetch( CBS.GAP_DISTANCES );
        UIC.Application_To_Device( CBS.GAP_DISTANCES );

    when CBS.GAP_MULTIPLIER =>
        CBC.Fetch( CBS.GAP_MULTIPLIER );
        UIC.Application_To_Device( CBS.GAP_MULTIPLIER );

    when CBS.GAP_MULTIPLIERS =>
        CBC.Fetch( CBS.GAP_MULTIPLIERS );
        UIC.Application_To_Device( CBS.GAP_MULTIPLIERS );

    when CBS.CONVOY =>
        UIC.Device_To_Application( DBS.CONVOY_NAME );
        DBC.Application_To_Device( ENTITY => DBS.CONVOY,
                                   STATUS => APP.STEADY );

        loop
            CBC.Fetch( CBS.CONVOY_PART );
            DBC.Application_To_Device( DBS.ELEMENT );
            exit when CBC.Signal = CBS.COMPLETE;
        end loop;
        CBC.Fetch( CBS.CONVOY_PARAMETERS );
        DBC.Application_To_Device( DBS.PARAMETERS );
        DBC.Application_To_Device( ENTITY => DBS.CONVOY,
                                   STATUS => APP.FINALIZE );

    when others => null;
end case;
end case;

when APP.ASSET_MANAGER =>
    case User_Selection.OPERATION is
        when APP.CONSTRUCT =>
            case User_Selection.ASSET_MANAGER_ENTITY is
                when AMS.MODEL =>
                    UIC.Device_To_Application( AMS.MODEL );
                    AMC.Construct( ENTITY => AMS.MODEL,
                                   SURROGATE => APP.USER_INTERFACE );
                    DBC.Application_To_Device( DBS.MODEL );

                when others => null;
            end case;

        when APP.DESTRUCT =>
            case User_Selection.ASSET_MANAGER_ENTITY is
                when AMS.MODEL =>
                    UIC.Device_To_Application( ENTITY => AMS.MODEL,
                                                KIND => UIC.KEY );
            end case;
        end case;
    end case;

```

```

        AMC.Destruct( AMS.MODEL );
        DBC.Application_To_Device( DBS.MODEL_ID );

        when others => null;
    end case;

    when APP.FETCH =>
        case User_Selection.ASSET_MANAGER_ENTITY is
            when AMS.MODEL =>
                UIC.Device_To_Application( ENTITY => AMS.MODEL,
                                           KIND => UIC.KEY );
                AMC.Fetch( AMS.MODEL );
                UIC.Application_To_Device( AMS.MODEL );

            when others => null;
        end case;
    end case;

    when APP.MAPPER =>
        case User_Selection.OPERATION is
            when APP.CONSTRUCT =>
                case User_Selection.MAPPER_ENTITY.ENTITY is
                    when MPS.VERTEX =>
                        UIC.Device_To_Application( MPS.VERTEX );
                        MPC.Construct( ENTITY => MPS.VERTEX,
                                     SOURCE => APP.USER_INTERFACE );

                    when MPS.ARC =>
                        UIC.Device_To_Application( MPS.ARC );
                        MPC.Construct( ENTITY => MPS.ARC,
                                     SOURCE => APP.USER_INTERFACE );

                    when MPS.CONSTRAINTS =>
                        CBC.Fetch( CBS.CONSTRAINTS );
                        MPC.Construct( MPS.CONSTRAINTS );
                        MPC.Fetch( MPS.CONSTRAINTS );
                        UIC.Application_To_Device( MPS.ROUTE );

                    when others => null;
                end case;
            when APP.DESTRUCT =>
                case User_Selection.MAPPER_ENTITY.ENTITY is
                    when MPS.VERTEX =>
                        UIC.Device_To_Application( ENTITY => MPS.VERTEX,
                                                  KIND => UIC.KEY );
                        MPC.Destruct( MPS.VERTEX );

                    when MPS.ARC =>
                        UIC.Device_To_Application( ENTITY => MPS.ARC,
                                                  KIND => UIC.KEY );
                        MPC.Destruct( MPS.ARC );

                    when MPS.CONSTRAINTS =>
                        MPC.Destruct( MPS.CONSTRAINTS );

                    when MPS.MAP =>
                        MPC.Destruct( MPS.Map );
                        DBC.Application_To_Device( ENTITY => DBS.MAP,

```

```

STATUS => APP.FINALIZE );

    when others => null;
end case;

when APP.FETCH =>
    case User_Selection.MAPPER_ENTITY.ENTITY is
        when MPS.ARC =>
            UIC.Device_To_Application( ENTITY => MPS.ARC,
                                         KIND => UIC.KEY );
            MPC.Fetch( MPS.ARC );
            UIC.Application_To_Device( MPS.ARC );

        when MPS.ROUTE =>
            UIC.Device_To_Application( MPS.ROUTE );
            CBC.Fetch( CBS.CONSTRAINTS );
            case User_Selection.MAPPER_ENTITY.FEATURE is
                when MPS.BEST =>
                    MPC.Fetch( MPS.BEST );
                when MPS.SATISFICE =>
                    null;
            end case;
            UIC.Application_To_Device( MPS.Route );

        when MPS.MAP =>
            UIC.Device_To_Application( DBS.MAP_NAME );
            DBC.Application_To_Device( ENTITY => DBS.MAP,
                                       STATUS => APP.STEADY );

            MPC.Construct( MPS.VERTICES );
            loop
                MPC.Fetch( MPS.VERTICES );
                exit when MPC.Signal = MPS.COMPLETE;
                DBC.Application_To_Device( DBS.VERTEX );
            end loop;

            MPC.Construct( MPS.ARCS );
            loop
                MPC.Fetch( MPS.ARCS );
                exit when MPC.Signal = MPS.COMPLETE;
                DBC.Application_To_Device( DBS.ARC );
            end loop;

            UIC.Device_To_Application( DBS.LOGICAL_ID );
            DBC.Application_To_Device( DBS.LOGICAL_ID );
            DBC.Application_To_Device( ENTITY => DBS.MAP,
                                       STATUS => APP.FINALIZE );

        when others => null;
    end case;
end case;

when APP.MARCH_TABLE =>
    case User_Selection.OPERATION is
        when APP.CONSTRUCT => null;

        when APP.DESTRUCT => null;

        when APP.FETCH =>

```



```

case User_Selection.MARCH_TABLE_ENTITY.ENTITY is
when MTS.MARCH_TABLE =>
    UIC.Device_To_Application( MTS.MARCH_TABLE );
    CBC.Fetch( CBS.LENGTH );
    CBC.Fetch( CBS.SPEED );
    MTC.Destruct( MTS.MARCH_TABLE );
case User_Selection.MARCH_TABLE_ENTITY.FEATURE
is
    when MTS.FORWARD =>
        MTC.Construct( ENTITY => MTS.MARCH_TABLE,
            FEATURE => MTS.FORWARD );

    when MTS.BACKWARD =>
        MTC.Construct( ENTITY => MTS.MARCH_TABLE,
            FEATURE => MTS.BACKWARD );

end case;
if MTC.Signal = MTS.NORMAL then
    MTC.Fetch( MTS.MARCH_TABLE );
    UIC.Application_To_Device( MTS.MARCH_TABLE );
else
    UIC.Application_To_Device(
        ( SUBSYSTEM => APP.MARCH_TABLE,
          MARCH_TABLE_ERROR =>
            ( ENTITY => MTS.MARCH_TABLE,
              STATUS => MTS.INVALID ) ) );
end if;

    when others => null;
end case;
end case;
end case;
end loop;

-- Finalization code to close some List files opened during initialization.
DBC.Application_To_Device( ENTITY => DBS.MODEL_LIST,
    STATUS => Program_State );
DBC.Application_To_Device( ENTITY => DBS.VEHICLE_LIST,
    STATUS => Program_State );
DBC.Application_To_Device( ENTITY => DBS.COMBINATION_LIST,
    STATUS => Program_State );
DBC.Application_To_Device( ENTITY => DBS.MAP_LIST,
    STATUS => Program_State );
DBC.Application_To_Device( ENTITY => DBS.CONVOY_LIST,
    STATUS => Program_State );
end Executive;

```

H.2 Application_Signatures

```
package Application_Signatures is

    type Application_Aggregate is ( USER_INTERFACE, DATA_BASE,
        CONVOY_BUILDER, MARCH_TABLE, MAPPER, ASSET_MANAGER );

    subtype Surrogate_Type is Application_Aggregate
        range USER_INTERFACE .. DATA_BASE;

    subtype Subsystem_Type is Application_Aggregate
        range CONVOY_BUILDER .. ASSET_MANAGER;

    type Status_Type is ( INITIALIZE, STEADY, FINALIZE );

    type Operation_Type is ( CONSTRUCT, DESTRUCT, FETCH );

end Application_Signatures;
```

H.3 Asset Manager Subsystem

H.3.1 Asset Manager Controller (Specification)

```
with Application_Signatures;
with Asset_Manager_Signatures;
package Asset_Manager_Controller is

    package AMS renames Asset_Manager_Signatures;
    package APP renames Application_Signatures;

    procedure Construct( ENTITY : in AMS.Entity_Type;
                        SURROGATE : in APP.Surrogate_Type );

    procedure Destruct( ENTITY : in AMS.Entity_Type );

    procedure Fetch( ENTITY : in AMS.Entity_Type );

    procedure Fetch( ENTITY : in AMS.Entity_Type;
                    AS_TYPE : in AMS.Entity_Type );

    function Signal return AMS.Status_Type;

end Asset_Manager_Controller;
```

H.3.2 Asset Manager Controller (Body)

```
with Asset_Manager_Imports;
with Asset_Manager_Exports;
with Vehicle_Types;
with Vehicle_Manager;
with Default;
package body Asset_Manager_Controller is

  -- renaming declaration for package abbreviations

  package AMI renames Asset_Manager_Imports;
  package AME renames Asset_Manager_Exports;
  package VM renames Vehicle_Manager;
  package VT renames Vehicle_Types;

  use Asset_Manager_Signatures;

  -----

  -- local variables and subprograms declared here

  INTERNAL_STATE : AMS.Status_Type;

  -- End declarations and code for internal subprograms
  -----
  -- Begin code for subprograms declared in package specification

  procedure Construct( ENTITY : in AMS.Entity_Type;
                      SURROGATE : in APP.Surrogate_Type ) is
  begin
    case ENTITY is
      when MODEL =>
        VM.Construct( MODEL_OR_VEHICLE => AMI.Vehicle( SURROGATE ),
                      KIND_OF_VEHICLE => VT.GENERAL );
        AME.Vehicle := AMI.Vehicle( SURROGATE );

      when SPECIFIC_VEHICLE =>
        VM.Construct( MODEL_OR_VEHICLE => AMI.Vehicle( SURROGATE ),
                      KIND_OF_VEHICLE => VT.SPECIFIC );
        AME.Vehicle := AMI.Vehicle( SURROGATE );

      when VEHICLE_COMBINATION =>
        VM.Construct( AMI.Combination( SURROGATE ) );
        AME.Vehicle_Combination := AMI.Combination( SURROGATE );

      when others => null;
    end case;
  end Construct;

  -----

  procedure Destruct( ENTITY : in AMS.Entity_Type ) is
  begin
    case ENTITY is
      when MODEL =>
        VM.Destruct( AMI.Model_Or_Combo_ID );
    end case;
  end Destruct;

  -----
```

```

when MODELS =>
    VM.Destruct( VT.General );

when SPECIFIC_VEHICLE =>
    VM.Destruct( AMI.Vehicle_ID );

when VEHICLES =>
    VM.Destruct( VT.Specific );

when VEHICLE_COMBINATION =>
    VM.Destruct( AMI.Model_Or_Combo_ID );

when COMBINATIONS =>
    VM.Destruct;

end case;
end Destruct;

```

```

procedure Fetch( ENTITY : in AMS.Entity_Type ) is
    Done : Boolean;
begin
    case ENTITY is
        when MODEL =>
            VM.Fetch( MODEL_ID => AMI.Model_Or_Combo_ID,
                      MODEL => AME.Vehicle );

        when MODELS =>
            VM.Fetch( KIND => VT.GENERAL,
                      VEHICLE => AME.Vehicle,
                      LAST => Done );
            if Done then
                INTERNAL_STATE := COMPLETE;
            else
                INTERNAL_STATE := INCOMPLETE;
            end if;

        when SPECIFIC_VEHICLE =>
            VM.Fetch( VEHICLE_ID => AMI.Vehicle_ID,
                      VEHICLE => AME.Vehicle );

        when VEHICLES =>
            VM.Fetch( KIND => VT.SPECIFIC,
                      VEHICLE => AME.Vehicle,
                      LAST => Done );
            if Done then
                INTERNAL_STATE := COMPLETE;
            else
                INTERNAL_STATE := INCOMPLETE;
            end if;

        when VEHICLE_COMBINATION =>
            VM.Fetch( COMBINATION_ID => AMI.Model_Or_Combo_ID,
                      COMBINATION => AME.Vehicle_Combination );

        when COMBINATIONS =>
            VM.Fetch( COMBINATION => AME.Vehicle_Combination,

```

```

        LAST => Done );
    if Done then
        INTERNAL_STATE := COMPLETE;
    else
        INTERNAL_STATE := INCOMPLETE;
    end if;

    when others => null;
end case;
end Fetch;

```

```

procedure Fetch( ENTITY : in AMS.Entity_Type;
                 AS_TYPE : in AMS.Entity_Type ) is
    Temp_Vehicle : VT.Vehicle_Type;
    Temp_Config : VT.Configured_Vehicle;
    Temp_Combo : VT.Vehicle_Combination( VT.Single );
begin
    case ENTITY is
        when MODEL =>
            case AS_TYPE is
                when VEHICLE_COMBINATION =>
                    VM.Fetch( MODEL_ID => AMI.Model_Or_Combo_ID,
                           MODEL => Temp_Vehicle );
                    Temp_Combo.Total := Temp_Vehicle.Properties;
                    Temp_Config.Vehicle := Temp_Vehicle;
                    Temp_Combo.Prime := Temp_Config;
                    AME.Vehicle_Combination := Temp_Combo;

                    when others => INTERNAL_STATE := INVALID;
            end case;

        when SPECIFIC_VEHICLE =>
            case AS_TYPE is
                when VEHICLE_COMBINATION =>
                    VM.Fetch( MODEL_ID => AMI.Model_Or_Combo_ID,
                           MODEL => Temp_Vehicle );
                    Temp_Combo.Total := Temp_Vehicle.Properties;
                    Temp_Config.Vehicle := Temp_Vehicle;
                    Temp_Combo.Prime := Temp_Config;
                    AME.Vehicle_Combination := Temp_Combo;

                    when others => INTERNAL_STATE := INVALID;
            end case;

        when others => INTERNAL_STATE := INVALID;
    end case;
end Fetch;

```

```

function Signal return AMS.Status_Type is
    Status : AMS.Status_Type := INTERNAL_STATE;
begin
    INTERNAL_STATE := NORMAL;
    return Status;
end Signal;

```

```
--*****  
-- Package initialization code!  
begin  
    INTERNAL_STATE := INITIALIZED;  
end Asset_Manager_Controller;
```

H.3.3 Asset Manager Signatures

```
package Asset_Manager_Signatures is

  type Entity_Type is ( MODEL, MODELS, SPECIFIC_VEHICLE, VEHICLES,
                        VEHICLE_COMBINATION, COMBINATIONS );

  type Status_Type is (INITIALIZED, INCOMPLETE, COMPLETE, INVALID, NORMAL);

  type Error_Type is record
    STATUS : Status_Type;
    ENTITY : Entity_Type;
  end record;

end Asset_Manager_Signatures;
```

H.3.4 Asset Manager Imports (Specification)

```
with Vehicle_Types;
with Application_Signatures;
package Asset_Manager_Imports is

  package APP renames Application_Signatures;
  package VT renames Vehicle_Types;

  function Model_Or_Combo_ID return VT.Model_Type;

  function Vehicle_ID return VT.Specific_Vehicle_Id;

  function Vehicle( SOURCE : in APP.Surrogate_Type )
    return VT.Vehicle_Type;

  function Combination( SOURCE : in APP.Surrogate_Type )
    return VT.Vehicle_Combination;

end Asset_Manager_Imports;
```


H.3.5 Asset Manager Imports (Body)

```
with User_Interface_Exports;
with Data_Base_Exports;
package body Asset_Manager_Imports is

    package UIE renames User_Interface_Exports;
    package DBE renames Data_Base_Exports;

    function Model_Or_Combo_ID return VT.Model_Type is
    begin
        return UIE.Model_ID;
    end Model_Or_Combo_ID;

    function Vehicle_ID return VT.Specific_Vehicle_Id is
    begin
        return UIE.Specific_Vehicle_Id;
    end Vehicle_ID;

    function Vehicle( SOURCE : in APP.Surrogate_Type )
                                return VT.Vehicle_Type is
    begin
        case SOURCE is
            when APP.USER_INTERFACE =>
                return UIE.Vehicle;

            when APP.DATA_BASE =>
                return DBE.Vehicle;
        end case;
    end Vehicle;

    function Combination( SOURCE : in APP.Surrogate_Type )
                                return VT.Vehicle_Combination is
    begin
        case SOURCE is
            when APP.USER_INTERFACE =>
                raise Program_Error;
                -- return UIE.Combination;
            when APP.DATA_BASE =>
                return DBE.Combination;
        end case;
    end Combination;

end Asset_Manager_Imports;
```

H.3.6 Asset Manager Exports

```
with Vehicle_Types;
package Asset_Manager_Exports is

    Vehicle : Vehicle_Types.Vehicle_Type;
    Vehicle_Combination : Vehicle_Types.Vehicle_Combination;

end Asset_Manager_Exports;
```

H.3.7 Vehicle Manager (Specification)

```
with Vehicle_Types;
package Vehicle_Manager is

    package VT renames Vehicle_Types;

    procedure Construct( MODEL_OR_VEHICLE : in VT.Vehicle_Type;
                        KIND_OF_VEHICLE : in VT.Vehicle_State );

    procedure Construct( COMBINATION : in VT.Vehicle_Combination );

    procedure Destruct( MODEL_OR_COMBO_ID : in VT.Model_Type );

    procedure Destruct( VEHICLE_WITH_ID : in VT.Specific_Vehicle_ID );

    procedure Destruct( KIND : in VT.Vehicle_State );
    -- Destroys entire contents of list specified by Kind

    procedure Destruct;
    -- Destroys entire contents of Combinations data

    procedure Fetch( MODEL_ID : in VT.Model_Type;
                   MODEL : out VT.Vehicle_Type );

    procedure Fetch( VEHICLE_ID : in VT.Specific_Vehicle_ID;
                   VEHICLE : out VT.Vehicle_Type );

    procedure Fetch( COMBINATION_ID : in VT.Model_Type;
                   COMBINATION : out VT.Vehicle_Combination );

    procedure Fetch( KIND : in VT.Vehicle_State;
                   VEHICLE : out VT.Vehicle_Type;
                   LAST : out Boolean );

    procedure Fetch( COMBINATION : out VT.Vehicle_Combination;
                   LAST : out Boolean );

    Not_Found : exception; -- raised if Fetch with an ID finds No Match

end Vehicle_Manager;
```

H.3.8 Vehicle Manager (Body)

```
with Ring_Sequential_Unbounded_Managed_Iterator;
with Default;
package body Vehicle_Manager is

    package Vehicle_Storage is new Ring_Sequential_Unbounded_Managed_Iterator
        ( ITEM => VT.Vehicle_Type );

    package Combo_Storage is new Ring_Sequential_Unbounded_Managed_Iterator
        ( ITEM => VT.Vehicle_Combination );

    function "="( L, R : in VT.Model_Type ) return Boolean renames VT."=";
    function "="( L, R : in VT.Vehicle_State ) return Boolean renames VT."=";
    function "="( L, R : in VT.Specific_Vehicle_Id ) return Boolean
        renames VT."=";

    -----

    -- local variables and subprograms declared here

    Models : Vehicle_Storage.Ring;
    Vehicles : Vehicle_Storage.Ring;
    Combinations : Combo_Storage.Ring;

    Fetch_Active : Boolean :=False;

    -----

    procedure Find_Model( MODEL_ID : in VT.Model_Type;
        FOUND : out Boolean ) is
    begin
        FOUND := False;
        Vehicle_Storage.Mark( Models );
        loop
            if Vehicle_Storage.Top_Of( Models ).Properties.Model = MODEL_ID
        then
            Vehicle_Storage.Mark( Models );
            FOUND := True;
        else
            Vehicle_Storage.Rotate( The_Ring => Models,
                In_The_Direction => Vehicle_Storage.Forward );
            end if;
            exit when Vehicle_Storage.At_Mark( Models );
        end loop;
    end Find_Model;

    -----

    procedure Find_Vehicle( VEHICLE_ID : in VT.Specific_Vehicle_Id;
        FOUND : out Boolean ) is
    begin
        FOUND := False;
        Vehicle_Storage.Mark( Vehicles );
        loop
            if Vehicle_Storage.Top_Of( Vehicles ).Vehicle_Id = VEHICLE_ID then
                Vehicle_Storage.Mark( Vehicles );
                FOUND := True;
            end if;
        end loop;
    end Find_Vehicle;

    -----
```

```

    else
        Vehicle_Storage.Rotate( The_Ring => Vehicles,
                                In_The_Direction => Vehicle_Storage.Forward );
    end if;
    exit when Vehicle_Storage.At_Mark( Vehicles );
end loop;
end Find_Vehicle;

-----

procedure Find_Combo( COMBO_ID : in VT.Model_Type;
                      FOUND : out Boolean ) is
begin
    FOUND := False;
    Combo_Storage.Mark( Combinations );
    loop
        if Combo_Storage.Top_Of( Combinations ).Total.Model = COMBO_ID then
            Combo_Storage.Mark( Combinations );
            FOUND := True;
        else
            Combo_Storage.Rotate( The_Ring => Combinations,
                                  In_The_Direction => Combo_Storage.Forward );
        end if;
        exit when Combo_Storage.At_Mark( Combinations );
    end loop;
end Find_Combo;

-- End declarations and code for internal subprograms
--
-----
-- Begin code for subprograms declared in package specification

procedure Construct( MODEL_OR_VEHICLE : in VT.Vehicle_Type;
                     KIND_OF_VEHICLE : in VT.Vehicle_State ) is
    Old_Value : Boolean := False;
begin
    case KIND_OF_VEHICLE is
        when VT.GENERAL =>
            if not Vehicle_Storage.Is_Empty( Models ) then -- look for OLD
version
                Find_Model( MODEL_ID => MODEL_OR_VEHICLE.Properties.Model,
                            FOUND => Old_Value );
                if Old_Value then -- Remove OLD version before Inserting NEW
                    Vehicle_Storage.Rotate_To_Mark( Models );
                    Vehicle_Storage.Pop( Models );
                end if;
            end if;
            Vehicle_Storage.Insert( The_Item => MODEL_OR_VEHICLE,
                                   In_The_Ring => Models );

        when VT.SPECIFIC =>
            if not Vehicle_Storage.Is_Empty( Vehicles ) then -- look for OLD
data
                Find_Vehicle( VEHICLE_ID => MODEL_OR_VEHICLE.Vehicle_Id,
                              FOUND => Old_Value );
                if Old_Value then -- Remove OLD version before Inserting NEW
                    Vehicle_Storage.Rotate_To_Mark( Vehicles );
                    Vehicle_Storage.Pop( Vehicles );
                end if;
            end if;
        end case;
    end Construct;

```

```

        end if;
        Vehicle_Storage.Insert( The_Item => MODEL_OR_VEHICLE,
                                In_The_Ring => Vehicles );
    end case;
end Construct;

-----

procedure Construct( COMBINATION : in VT.Vehicle_Combination ) is
    Old_Value : Boolean := False;
begin
    if not Combo_Storage.Is_Empty( Combinations ) then -- look for OLD
version
        Find_Combo( COMBO_ID => COMBINATION.Total.Model,
                    FOUND => Old_Value );
        if Old_Value then -- Remove OLD version before Inserting NEW
            Combo_Storage.Rotate_To_Mark( Combinations );
            Combo_Storage.Pop( Combinations );
        end if;
    end if;
    Combo_Storage.Insert( The_Item => COMBINATION,
                          In_The_Ring => Combinations );
end Construct;

-----

procedure Destruct( MODEL_OR_COMBO_ID : in VT.Model_Type ) is
    Id_Found : Boolean := False;
begin
    if MODEL_OR_COMBO_ID( 9 ) = ' ' then -- This is a MODEL
        Find_Model( MODEL_ID => MODEL_OR_COMBO_ID,
                    FOUND => Id_Found );
        if Id_Found then -- Remove
            Vehicle_Storage.Rotate_To_Mark( Models );
            Vehicle_Storage.Pop( Models );
        end if;
    else -- This MUST be a COMBINATION
        Find_Combo( COMBO_ID => MODEL_OR_COMBO_ID,
                    FOUND => Id_Found );
        if Id_Found then -- Remove
            Combo_Storage.Rotate_To_Mark( Combinations );
            Combo_Storage.Pop( Combinations );
        end if;
    end if;
end Destruct;

-----

procedure Destruct( VEHICLE_WITH_ID : in VT.Specific_Vehicle_ID ) is
    Id_Found : Boolean := False;
begin
    Find_Vehicle( VEHICLE_ID => VEHICLE_WITH_ID,
                  FOUND => Id_Found );
    if Id_Found then -- Remove
        Vehicle_Storage.Rotate_To_Mark( Vehicles );
        Vehicle_Storage.Pop( Vehicles );
    end if;
end Destruct;

```

```

--*****

procedure Destruct( KIND : in VT.Vehicle_State ) is
begin
  case KIND is
    when VT.GENERAL =>
      Vehicle_Storage.Clear( Models );
    when VT.SPECIFIC =>
      Vehicle_Storage.Clear( Vehicles );
    end case;
  end Destruct;

--*****

procedure Destruct is
begin
  Combo_Storage.Clear( Combinations );
  end Destruct;

--*****

procedure Fetch( MODEL_ID : in VT.Model_Type;
                 MODEL : out VT.Vehicle_Type ) is
  Id_Found : Boolean := False;
begin
  if not Vehicle_Storage.Is_Empty( Models ) then
    Find_Model( MODEL_ID => MODEL_ID,
               FOUND => Id_Found );
    if Id_Found then
      Vehicle_Storage.Rotate_To_Mark( Models );
      MODEL := Vehicle_Storage.Top_Of( Models );
    else
      MODEL := Default.General_Vehicle;
      MODEL.Properties.Model := MODEL_ID;
    end if;
  else
    MODEL := Default.General_Vehicle;
    MODEL.Properties.Model := MODEL_ID;
  end if;
end Fetch;

--*****

procedure Fetch( VEHICLE_ID : in VT.Specific_Vehicle_ID;
                 VEHICLE : out VT.Vehicle_Type ) is
  Id_Found : Boolean := False;
begin
  if not Vehicle_Storage.Is_Empty( Vehicles ) then
    Find_Vehicle( VEHICLE_ID => VEHICLE_ID,
                 FOUND => Id_Found );
    if Id_Found then
      Vehicle_Storage.Rotate_To_Mark( Vehicles );
      VEHICLE := Vehicle_Storage.Top_Of( Vehicles );
    else
      raise Not_Found;
    end if;
  else
    raise Not_Found;
  end if;
end Fetch;

```

```

    end if;
end Fetch;

```

```

procedure Fetch( COMBINATION_ID : in VT.Model_Type;
                 COMBINATION : out VT.Vehicle_Combination ) is
    Id_Found : Boolean := False;
begin
    if not Combo_Storage.Is_Empty( Combinations ) then
        Find_Combo( COMBO_ID => COMBINATION_ID,
                   FOUND => Id_Found );
        if Id_Found then -- Remove OLD version before Inserting NEW
            Combo_Storage.Rotate_To_Mark( Combinations );
            COMBINATION := Combo_Storage.Top_Of( Combinations );
        else
            raise Not_Found;
        end if;
    else
        raise Not_Found;
    end if;
end Fetch;

```

```

procedure Fetch( KIND : in VT.Vehicle_State;
                 VEHICLE : out VT.Vehicle_Type;
                 LAST : out Boolean ) is
begin
    case KIND is
        when VT.GENERAL =>
            if not Fetch_Active then
                if Vehicle_Storage.Is_Empty( Models ) then
                    raise Not_Found;
                end if;
                Vehicle_Storage.Mark( Models ); -- This will be the LAST one
done!
                Fetch_Active := True;
            end if;
            Vehicle_Storage.Rotate( The_Ring => Models,
                                   In_The_Direction => Vehicle_Storage.Forward );
            VEHICLE := Vehicle_Storage.Top_Of( Models );
            if Vehicle_Storage.At_Mark( Models ) then -- At the LAST one?
                Fetch_Active := False;
                LAST := True;
            else
                LAST := False;
            end if;

        when VT.SPECIFIC =>
            if not Fetch_Active then
                if Vehicle_Storage.Is_Empty( Vehicles ) then
                    raise Not_Found;
                end if;
                Vehicle_Storage.Mark( Vehicles );
                Fetch_Active := True;
            end if;
            Vehicle_Storage.Rotate( The_Ring => Vehicles,
                                   In_The_Direction => Vehicle_Storage.Forward );
    end case;
end Fetch;

```

```

        VEHICLE := Vehicle_Storage.Top_Of( Vehicles );
        if Vehicle_Storage.At_Mark( Vehicles ) then
            Fetch_Active := False;
            LAST := True;
        else
            LAST := False;
        end if;

    end case;
end Fetch;

--*****

procedure Fetch( COMBINATION : out VT.Vehicle_Combination;
                 LAST : out Boolean ) is
begin
    if not Fetch_Active then
        if Combo_Storage.Is_Empty( Combinations ) then
            raise Not_Found;
        end if;
        Combo_Storage.Mark( Combinations );
        Fetch_Active := True;
    end if;
    Combo_Storage.Rotate( The_Ring => Combinations,
                        In_The_Direction => Combo_Storage.Forward );
    COMBINATION := Combo_Storage.Top_Of( Combinations );
    if Combo_Storage.At_Mark( Combinations ) then
        Fetch_Active := False;
        LAST := True;
    else
        LAST := False;
    end if;
end Fetch;

--*****

end Vehicle_Manager;

```


H.4 Data Base Surrogate

H.4.1 Data Base Controller (Specification)

```
with Data_Base_Signatures;  
with Application_Signatures;  
package Data_Base_Controller is  
  
    package APP renames Application_Signatures;  
    package DBS renames Data_Base_Signatures;  
  
    procedure Device_To_Application( ENTITY : in DBS.Entity_Type );  
  
    procedure Application_To_Device( ENTITY : in DBS.Entity_Type;  
                                     STATUS : in APP.Status_Type  
                                     := APP.STEADY );  
  
    function Signal return DBS.Status_Type;  
  
end Data_Base_Controller;
```

H.4.2 Data Base Controller (Body)

```
with Vehicle_Types;
with Data_Base_Imports;
with Data_Base_Exports;
with DB_Vehicle_IO;
with DB_Convoy_IO;
with DB_Map_IO;
package body Data_Base_Controller is

    package DBE renames Data_Base_Exports;
    package DBI renames Data_Base_Imports;

    -- local variables and subprograms declared here

    SIGNAL_STATE : DBS.Status_Type;
    Count_Read : Boolean := False;

    -- End declarations and code for internal subprograms
    --
    *****
    -- Begin code for subprograms declared in package specification

    procedure Device_To_Application( ENTITY : in DBS.Entity_Type ) is
    begin
        case ENTITY is
            when DBS.MAP_NAME =>
                begin
                    DB_Map_IO.Get_Map_From_List( DBE.Map_Name );
                exception
                    when DB_Map_IO.End_Of_File =>
                        SIGNAL_STATE := DBS.END_OF_FILE;
                    end;

            when DBS.VERTEX =>
                begin
                    if not Count_Read then
                        DBE.Records_In_File := DB_Map_IO.Number_Of_Vertices;
                        Count_Read := True;
                    else
                        DB_Map_IO.Get( DBE.Vertex_Data );
                    end if;
                exception
                    when DB_Map_IO.End_Of_File =>
                        SIGNAL_STATE := DBS.END_OF_FILE;
                        Count_Read := False;
                    end;

            when DBS.ARC =>
                begin
                    if not Count_Read then
                        DBE.Records_In_File := DB_Map_IO.Number_Of_Arcs;
                        Count_Read := True;
                    else
                        DB_Map_IO.Get( DBE.Arc_Data );
                    end if;
                exception
                    when DB_Map_IO.End_Of_File =>
                        SIGNAL_STATE := DBS.END_OF_FILE;
```

```

        Count_Read := False;
    end;

    when DBS.LOGICAL_ID =>
        DB_Map_IO.Get( DBE.Logical_Id_Value );

    when DBS.MODEL =>
        begin
            DB_Vehicle_IO.Get_Model( DBE.Vehicle );
        exception
            when DB_Vehicle_IO.End_Of_File =>
                SIGNAL_STATE := DBS.END_OF_FILE;
            end;

    when DBS.MODEL_ID =>
        DBE.Model_Id := DBE.Vehicle.Properties.Model;

    when DBS.SPECIFIC_VEHICLE =>
        begin
            DB_Vehicle_IO.Get_Vehicle( DBE.Vehicle );
        exception
            when DB_Vehicle_IO.End_Of_File =>
                SIGNAL_STATE := DBS.END_OF_FILE;
            end;

    when DBS.VEHICLE_ID =>
        DBE.Model_Id := DBE.Vehicle.Properties.Model;
        DBE.Vehicle_Id := DBE.Vehicle.Vehicle_Id;

    when DBS.VEHICLE_COMBINATION =>
        begin
            DB_Vehicle_IO.Get_Combination( DBE.Combination );
        exception
            when DB_Vehicle_IO.End_Of_File =>
                SIGNAL_STATE := DBS.END_OF_FILE;
            end;

    when DBS.COMBINATION_ID =>
        DBE.Model_Id := DBE.Combination.Total.Model;

    when DBS.CONVOY_NAME =>
        begin
            DB_Convoy_IO.Get_Convoy_From_List( DBE.Convoy_Name );
        exception
            when DB_Convoy_IO.End_Of_File =>
                SIGNAL_STATE := DBS.END_OF_FILE;
            end;

    when DBS.ELEMENT =>
        begin
            if not Count_Read then
                DBE.Records_In_File := DB_Convoy_IO.Number_Of_Elements;
                Count_Read := True;
            else
                DB_Convoy_IO.Get( DBE.Convoy_Part );
            end if;
        exception
            when DB_Convoy_IO.End_Of_File =>
                SIGNAL_STATE := DBS.END_OF_FILE;
        end;

```

```

        Count_Read := False;
    end;

    when DBS.PARAMETERS =>
        DB_Convoy_IO.Get( DBE.Convoy_Data );

    when others => null;
end case;
end Device_To_Location;

-----
procedure Application_To_Device( ENTITY : in DBS.Entity_Type;
                                STATUS : in APP.Status_Type
                                := APP.STEADY ) is
begin
    case ENTITY is
        when DBS.MAP_LIST =>
            case STATUS is
                when APP.INITIALIZE =>
                    DB_Map_IO.Open_List_File;
                    DBE.Records_In_File := DB_Map_IO.Number_Of_Maps;

                when APP.STEADY => null;

                when APP.FINALIZE =>
                    DB_Map_IO.Close_List_File;
            end case;

        when DBS.MAP =>
            case STATUS is
                when APP.INITIALIZE =>
                    begin
                        DB_Map_IO.Open_Map_Files( Name => DBI.Map_Name,
                                                  Mode => DB_Map_IO.Input );
                    exception
                        when DB_Map_IO.Invalid_Map_Name =>
                            SIGNAL_STATE := DBS.NOT_FOUND;
                    end;

                when APP.STEADY =>
                    DB_Map_IO.Open_Map_Files( Name => DBI.Map_Name,
                                              Mode => DB_Map_IO.Output );
                    DB_Map_IO.Put_Map_In_List( DBI.Map_Name );

                when APP.FINALIZE =>
                    DB_Map_IO.Close_Map_Files;
            end case;

        when DBS.MAP_NAME =>
            DB_Map_IO.Delete_Map_Files( DBI.Map_Name );
            DB_Map_IO.Remove_Map_From_List( DBI.Map_Name );

        when DBS.VERTEX =>
            DB_Map_IO.Put( DBI.Get_Vertex );

        when DBS.ARC =>
            DB_Map_IO.Put( DBI.Get_Arc );

        when DBS.LOGICAL_ID =>

```

```

DB_Map_IO.Put( DBI.Logical_Id_Value );

when DBS.MODEL_LIST =>
  case STATUS is
    when APP.INITIALIZE =>
      DB_Vehicle_IO.Open_Models_File;
      DBE.Records_In_File := DB_Vehicle_IO.Number_Of_Models;

    when APP.STEADY => null;

    when APP.FINALIZE =>
      DB_Vehicle_IO.Close_Models_File;
  end case;

when DBS.MODEL =>
  DB_Vehicle_IO.Put_Model( DBI.Get_Vehicle );

when DBS.MODEL_ID =>
  DB_Vehicle_IO.Remove_Model( DBI.Get_Vehicle );

when DBS.VEHICLE_LIST =>
  case STATUS is
    when APP.INITIALIZE =>
      DB_Vehicle_IO.Open_Vehicles_File;
      DBE.Records_In_File :=
        DB_Vehicle_IO.Number_Of_Vehicles;

    when APP.STEADY => null;

    when APP.FINALIZE =>
      DB_Vehicle_IO.Close_Vehicles_File;
  end case;

when DBS.SPECIFIC_VEHICLE =>
  DB_Vehicle_IO.Put_Vehicle( DBI.Get_Vehicle );

when DBS.VEHICLE_ID =>
  DB_Vehicle_IO.Remove_Vehicle( DBI.Get_Vehicle );

when DBS.COMBINATION_LIST =>
  case STATUS is
    when APP.INITIALIZE =>
      DB_Vehicle_IO.Open_Combinations_File;
      DBE.Records_In_File :=
        DB_Vehicle_IO.Number_Of_Combinations;

    when APP.STEADY => null;

    when APP.FINALIZE =>
      DB_Vehicle_IO.Close_Combinations_File;
  end case;

when DBS.VEHICLE_COMBINATION =>
  DB_Vehicle_IO.Put_Combination( DBI.Get_Combination );

when DBS.COMBINATION_ID =>
  DB_Vehicle_IO.Remove_Combination( DBI.Get_Combination );

when DBS.CONVOY_LIST =>

```

```

    case STATUS is
        when APP.INITIALIZE =>
            DB_Convoy_IO.Open_List_File;
            DBE.Records_In_File := DB_Convoy_IO.Number_Of_Convoys;

        when APP.STEADY => null;

        when APP.FINALIZE =>
            DB_Convoy_IO.Close_List_File;
    end case;

    when DBS.CONVOY =>
        case STATUS is
            when APP.INITIALIZE =>
                DB_Convoy_IO.Open_Convoy_Files( DBI.Convoy_Name,
                                                DB_Convoy_IO.Input );

            when APP.STEADY =>
                DB_Convoy_IO.Open_Convoy_Files( DBI.Convoy_Name,
                                                DB_Convoy_IO.Output );
                DB_Convoy_IO.Put_Convoy_In_List( DBI.Convoy_Name );

            when APP.FINALIZE =>
                DB_Convoy_IO.Close_Convoy_Files;
        end case;

    when DBS.CONVOY_NAME =>
        DB_Convoy_IO.Delete_Convoy_Files( DBI.Convoy_Name );
        DB_Convoy_IO.Remove_Convoy_From_List( DBI.Convoy_Name );

    when DBS.ELEMENT =>
        DB_Convoy_IO.Put( DBI.Get_Convoy_Part );

    when DBS.PARAMETERS =>
        DB_Convoy_IO.Put( DBI.Get_Convoy_Parameters );
    end case;
end Application_To_Device;

--*****
function Signal return DBS.Status_Type is
    Status : DBS.Status_Type := SIGNAL_STATE;
begin
    SIGNAL_STATE := DBS.NORMAL; -- RESET upon READ!
    return Status;
end Signal;

--*****
begin
    SIGNAL_STATE := DBS.INITIALIZED;
end Data_Base_Controller;

```

H.4.3 Data Base Signatures

```
package Data_Base_Signatures is

  type Entity_Type is ( MAP_LIST, MAP, MAP_NAME, VERTEX, ARC, LOGICAL_ID,
                        MODEL_LIST, MODEL, MODEL_ID, VEHICLE_LIST,
                        SPECIFIC_VEHICLE, VEHICLE_ID, COMBINATION_LIST,
                        VEHICLE_COMBINATION, COMBINATION_ID, CONVOY_LIST,
                        CONVOY, CONVOY_NAME, ELEMENT, PARAMETERS );

  type Status_Type is ( INITIALIZED, NOT_FOUND, END_OF_FILE, NORMAL );

  type Error_Type is record
    STATUS : Status_Type;
    ENTITY : Entity_Type;
  end record;

end Data_Base_Signatures;
```

H.4.4 Data Base Types

```
with Convoy_Builder_Types;
with Vehicle_Types;
with Measurement_Types;
package DB_Types is

    Max_Name_Length : constant Positive := 20;

    -- Data construct to store Convoy Vehicle and organization info.

    package CBT renames Convoy_Builder_Types;

    type Convoy_Element_Type( Level : CBT.Levels_Type := CBT.Vehicle ) is
    record
        case Level is
            when CBT.Levels_Type'Last =>
                Info : Vehicle_Types.Vehicle_Combination;

            when others => null; -- will be Internal_Org info. in future!
        end case;
    end record;

    -- Data constructs to store other Convoy dependent Parameters

    subtype Number_Of_Levels is Natural range 0 ..
        CBT.Levels_Type'Pos( CBT.Levels_Type'Last );

    type Fixed_Gap_Data is array ( Number_Of_Levels ) of
        Measurement_Types.Distance_Measurement;

    type Governed_Gap_Data is array ( Number_Of_Levels ) of
        Measurement_Types.Gap_Multiplier_Type;

    type Int_Orged_Levels is array ( Number_Of_Levels range <> ) of
        CBT.Levels_Type;

    type Convoy_Parameters_Type( Governed : Boolean := True;
        Org_Levels : Number_Of_Levels :=
            Number_Of_Levels'Last ) is
    record
        Average_Speed : Measurement_Types.Rate_Measurement;
        Org_Data : Int_Orged_Levels( 1 .. Org_Levels );
        case Governed is
            when True =>
                Governed_Gaps : Governed_Gap_Data;
            when False =>
                Fixed_Gaps : Fixed_Gap_Data;
        end case;
    end record;

end DB_Types;
```


H.4.5 Data Base Imports (Specification)

```
with Vehicle_Types;
with Map_Types;
with DB_Types;
package Data_Base_Imports is

    function Get_Vehicle return Vehicle_Types.Vehicle_Type;

    function Get_Combination return Vehicle_Types.Vehicle_Combination;

    function Map_Name return String;

    function Get_Vertex return Map_Types.Vertex_Type;

    function Get_Arc return Map_Types.Arc_Type;

    function Logical_Id_Value return Natural;

    function Convoy_Name return String;

    function Get_Convoy_Part return DB_Types.Convoy_Element_Type;

    function Get_Convoy_Parameters return DB_Types.Convoy_Parameters_Type;

end Data_Base_Imports;
```

H.4.6 Data Base Imports (Body)

```
with Asset_Manager_Exports;
with User_Interface_Exports;
with Convoy_Builder_Exports;
with Mapper_Exports;
package body Data_Base_Imports is

    package AME renames Asset_Manager_Exports;
    package CBE renames Convoy_Builder_Exports;
    package MPE renames Mapper_Exports;
    package UIE renames User_Interface_Exports;

    function Get_Vehicle return Vehicle_Types.Vehicle_Type is
    begin
        return AME.Vehicle;
    end Get_Vehicle;

    function Get_Combination return Vehicle_Types.Vehicle_Combination is
    begin
        return AME.Vehicle_Combination;
    end Get_Combination;

    function Map_Name return String is
    begin
        return UIE.Map_Name;
    end Map_Name;
```

```

function Get_Vertex return Map_Types.Vertex_Type is
begin
    return MPE.Vertex;
end Get_Vertex;

function Get_Arc return Map_Types.Arc_Type is
begin
    return MPE.Arc;
end Get_Arc;

function Logical_Id_Value return Natural is
begin
    return UIE.Vertex_Id;
end Logical_Id_Value;

function Convoy_Name return String is
begin
    return UIE.Convoy_Name;
end Convoy_Name;

function Get_Convoy_Part return DB_Types.Convoy_Element_Type is
begin
    return CBE.Convoy_Part;
end Get_Convoy_Part;

function Get_Convoy_Parameters return DB_Types.Convoy_Parameters_Type is
begin
    return CBE.Parameters;
end Get_Convoy_Parameters;

end Data_Base_Imports;

```

H.4.7 Data Base Exports

```

with Vehicle_Types;
with Map_Types;
with Default;
with DB_Types;
package Data_Base_Exports is

    Records_In_File : Natural;

    Model_Id : Vehicle_Types.Model_Type;
    Vehicle_Id : Vehicle_Types.Specific_Vehicle_Id;
    Vehicle : Vehicle_Types.Vehicle_Type := Default.General_Vehicle;
    Combination : Vehicle_Types.Vehicle_Combination;

    Convoy_Name : String( 1 .. DB_Types.Max_Name_Length );
    Convoy_Part : DB_Types.Convoy_Element_Type;
    Convoy_Data : DB_Types.Convoy_Parameters_Type;

    Map_Name : String( 1 .. DB_Types.Max_Name_Length );
    Vertex_Data : Map_Types.Vertex_Type;
    Arc_Data : Map_Types.Arc_Type;
    Logical_Id_Value : Natural;

end Data_Base_Exports;

```

H.4.8 Vehicle_IO Handler (Specification)

```
with Vehicle_Types ;
package DB_Vehicle_IO is

    procedure Open_Models_File;

    function Number_Of_Models return Natural;

    procedure Put_Model( Vehicle : in Vehicle_Types.Vehicle_Type );

    procedure Get_Model( Vehicle : out Vehicle_Types.Vehicle_Type );

    procedure Remove_Model( Vehicle : in Vehicle_Types.Vehicle_Type );

    procedure Close_Models_File;

    procedure Open_Vehicles_File;

    function Number_Of_Vehicles return Natural;

    procedure Put_Vehicle( Vehicle : in Vehicle_Types.Vehicle_Type );

    procedure Get_Vehicle( Vehicle : out Vehicle_Types.Vehicle_Type );

    procedure Remove_Vehicle( Vehicle : in Vehicle_Types.Vehicle_Type );

    procedure Close_Vehicles_File;

    procedure Open_Combinations_File;

    function Number_Of_Combinations return Natural;

    procedure Put_Combination( Combo : in Vehicle_Types.Vehicle_Combination);

    procedure Get_Combination(Combo : out Vehicle_Types.Vehicle_Combination);

    procedure Remove_Combination( Combo : in
                                   Vehicle_Types.Vehicle_Combination );

    procedure Close_Combinations_File;

    End_Of_File : exception; -- raised by any Get that attempts a read past EOF

end DB_Vehicle_IO;
```

H.4.9 Vehicle_IO Handler (Body)

```
with Direct_IO;
with File_Data;
package body DB_Vehicle_IO is

    package Veh_IO is new Direct_IO( Vehicle_Types.Vehicle_Type );
    package Combo_IO is new Direct_IO( Vehicle_Types.Vehicle_Combination );

    function "="( L, R : in Vehicle_Types.Model_Type ) return Boolean
        renames Vehicle_Types."=";

    function "="( L, R : in Vehicle_Types.Specific_Vehicle_Id ) return Boolean
        renames Vehicle_Types."=";

    function "="( L, R : in Vehicle_Types.Vehicle_Combination ) return Boolean
        renames Vehicle_Types."=";

    Model_File : Veh_IO.File_Type;
    Vehicle_File : Veh_IO.File_Type;
    Combination_File : Combo_IO.File_Type;

    -----

    procedure Open_Models_File is
    begin
        Veh_IO.Open( File => Model_File,
                     Mode => Veh_IO.Inout_File,
                     Name => File_Data.Path & "Models.LST" );
    exception
        when Veh_IO.Name_Error =>
            Veh_IO.Create( File => Model_File,
                          Mode => Veh_IO.Inout_File,
                          Name => File_Data.Path & "Models.LST" );
        when others => raise;
    end Open_Models_File;

    function Number_Of_Models return Natural is
    begin
        return Natural( Veh_IO.Size( Model_File ) );
    exception
        when others => return 0;
    end Number_Of_Models;

    procedure Put_Model( Vehicle : in Vehicle_Types.Vehicle_Type ) is
        DB_Data : Vehicle_Types.Vehicle_Type;
        Written : Boolean := False;
    begin
        for Index in 1..Veh_IO.Size( Model_File ) loop
            Veh_IO.Read( File => Model_File,
                         Item => DB_Data,
                         From => Index );
            if DB_Data.Properties.Model = Vehicle.Properties.Model then
                Veh_IO.Write( File => Model_File,
                              Item => Vehicle,
                              To => Index );
                Written := True;
            exit;
        end if;
    end Put_Model;
```

```

        end loop;
        if not Written then
            Veh_IO.Write( File => Model_File,
                          Item => Vehicle );
        end if;
    end Put_Model;

    procedure Get_Model( Vehicle : out Vehicle_Types.Vehicle_Type ) is
    begin
        Veh_IO.Read( File => Model_File,
                     Item => Vehicle );
    exception
        when Veh_IO.End_Error => raise End_Of_File;
    end Get_Model;

    procedure Remove_Model( Vehicle : in Vehicle_Types.Vehicle_Type ) is
        DB_Data : Vehicle_Types.Vehicle_Type;
        Temp_File : Veh_IO.File_Type;
    begin
        Veh_IO.Create( File => Temp_File,
                       Mode => Veh_IO.Inout_File,
                       Name => File_Data.Path & "Models.TLST" );
        for Index in 1..Veh_IO.Size( Model_File ) loop
            Veh_IO.Read( File => Model_File,
                         Item => DB_Data,
                         From => Index );
            if DB_Data.Properties.Model /= Vehicle.Properties.Model then
                Veh_IO.Write( File => Temp_File,
                              Item => DB_Data );
            end if;
        end loop;
        Veh_IO.Delete( Model_File );
        Veh_IO.Create( File => Model_File,
                       Mode => Veh_IO.Inout_File,
                       Name => File_Data.Path & "Models.LST" );
        for Index in 1..Veh_IO.Size( Temp_File ) loop
            Veh_IO.Read( File => Temp_File,
                         Item => DB_Data,
                         From => Index );
            Veh_IO.Write( File => Model_File,
                          Item => DB_Data );
        end loop;
        Veh_IO.Delete( Temp_File );
    end Remove_Model;

    procedure Close_Models_File is
    begin
        Veh_IO.Close( Model_File );
    end Close_Models_File;

    -----

    procedure Open_Vehicles_File is
    begin
        Veh_IO.Open( File => Vehicle_File,
                     Mode => Veh_IO.Inout_File,
                     Name => File_Data.Path & "Vehicles.LST" );
    exception
        when Veh_IO.Name_Error =>

```

```

        Veh_IO.Create( File => Vehicle_File,
                       Mode => Veh_IO.Inout_File,
                       Name => File_Data.Path & "Vehicles.LST" );
    when others => raise;
end Open_Vehicles_File;

function Number_Of_Vehicles return Natural is
begin
    return Natural( Veh_IO.Size( Vehicle_File ) );
exception
    when others => return 0;
end Number_Of_Vehicles;

procedure Put_Vehicle( Vehicle : in Vehicle_Types.Vehicle_Type ) is
    DB_Data : Vehicle_Types.Vehicle_Type;
    Written : Boolean := False;
begin
    for Index in 1..Veh_IO.Size( Vehicle_File ) loop
        Veh_IO.Read( File => Vehicle_File,
                     Item => DB_Data,
                     From => Index );
        if DB_Data.Vehicle_Id = Vehicle.Vehicle_Id then
            Veh_IO.Write( File => Vehicle_File,
                          Item => Vehicle,
                          To => Index );
            Written := True;
            exit;
        end if;
    end loop;
    if not Written then
        Veh_IO.Write( File => Vehicle_File,
                      Item => Vehicle );
    end if;
end Put_Vehicle;

procedure Get_Vehicle( Vehicle : out Vehicle_Types.Vehicle_Type ) is
begin
    Veh_IO.Read( File => Vehicle_File,
                 Item => Vehicle );
exception
    when Veh_IO.End_Error => raise End_Of_File;
end Get_Vehicle;

procedure Remove_Vehicle( Vehicle : in Vehicle_Types.Vehicle_Type ) is
    DB_Data : Vehicle_Types.Vehicle_Type;
    Temp_File : Veh_IO.File_Type;
begin
    Veh_IO.Create( File => Temp_File,
                   Mode => Veh_IO.Inout_File,
                   Name => File_Data.Path & "Vehicles.TLST" );
    for Index in 1..Veh_IO.Size( Vehicle_File ) loop
        Veh_IO.Read( File => Vehicle_File,
                     Item => DB_Data,
                     From => Index );
        if DB_Data.Vehicle_Id /= Vehicle.Vehicle_Id then
            Veh_IO.Write( File => Temp_File,
                          Item => DB_Data );
        end if;
    end loop;

```

```

Veh_IO.Delete( Vehicle_File );
Veh_IO.Create( File => Vehicle_File,
               Mode => Veh_IO.Inout_File,
               Name => File_Data.Path & "Vehicles.LST" );
for Index in 1..Veh_IO.Size( Temp_File ) loop
    Veh_IO.Read( File => Temp_File,
                Item => DB_Data,
                From => Index );
    Veh_IO.Write( File => Vehicle_File,
                 Item => DB_Data );
end loop;
Veh_IO.Delete( Temp_File );
end Remove_Vehicle;

```

```

procedure Close_Vehicles_File is
begin
    Veh_IO.Close( Vehicle_File );
end Close_Vehicles_File;

```

```

procedure Open_Combinations_File is
begin
    Combo_IO.Open( File => Combination_File,
                  Mode => Combo_IO.Inout_File,
                  Name => File_Data.Path & "Combinations.LST" );
exception
when Combo_IO.Name_Error =>
    Combo_IO.Create( File => Combination_File,
                    Mode => Combo_IO.Inout_File,
                    Name => File_Data.Path & "Combinations.LST" );
when others => raise;
end Open_Combinations_File;

```

```

function Number_Of_Combinations return Natural is
begin
    return Natural( Combo_IO.Size( Combination_File ) );
exception
when others => return 0;
end Number_Of_Combinations;

```

```

procedure Put_Combination
    ( Combo : in Vehicle_Types.Vehicle_Combination ) is
    DB_Data : Vehicle_Types.Vehicle_Combination;
    Written : Boolean := False;
begin
    for Index in 1..Combo_IO.Size( Combination_File ) loop
        Combo_IO.Read( File => Combination_File,
                       Item => DB_Data,
                       From => Index );
        if DB_Data = Combo then
            Combo_IO.Write( File => Combination_File,
                           Item => Combo,
                           To => Index );
            Written := True;
            exit;
        end if;
    end loop;
    if not Written then

```

```

        Combo_IO.Write( File => Combination_File,
                        Item => Combo );
    end if;
end Put_Combination;

procedure Get_Combination
    ( Combo : out Vehicle_Types.Vehicle_Combination ) is
begin
    Combo_IO.Read( File => Combination_File,
                  Item => Combo );
exception
    when Combo_IO.End_Error => raise End_Of_File;
end Get_Combination;

procedure Remove_Combination
    ( Combo : in Vehicle_Types.Vehicle_Combination ) is
    DB_Data : Vehicle_Types.Vehicle_Combination;
    Temp_File : Combo_IO.File_Type;
begin
    Combo_IO.Create( File => Temp_File,
                    Mode => Combo_IO.Inout_File,
                    Name => File_Data.Path & "Combinations.TLST" );
    for Index in 1..Combo_IO.Size( Combination_File ) loop
        Combo_IO.Read( File => Combination_File,
                      Item => DB_Data,
                      From => Index );
        if DB_Data /= Combo then
            Combo_IO.Write( File => Temp_File,
                           Item => DB_Data );
        end if;
    end loop;
    Combo_IO.Delete( Combination_File );
    Combo_IO.Create( File => Combination_File,
                    Mode => Combo_IO.Inout_File,
                    Name => File_Data.Path & "Combinations.LST" );
    for Index in 1..Combo_IO.Size( Temp_File ) loop
        Combo_IO.Read( File => Temp_File,
                      Item => DB_Data,
                      From => Index );
        Combo_IO.Write( File => Combination_File,
                       Item => DB_Data );
    end loop;
    Combo_IO.Delete( Temp_File );
end Remove_Combination;

procedure Close_Combinations_File is
begin
    Combo_IO.Close( Combination_File );
end Close_Combinations_File;

end DB_Vehicle_IO;

```


REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS None		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-94-TR-8			5. MONITORING ORGANIZATION REPORT NUMBER(S) ESC-TR-94-008		
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute		6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office		
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213			7b. ADDRESS (city, state, and zip code) HQ ESC/ENS 5 Eglin Street Hanscom AFB, MA 01731-2116		
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office		8b. OFFICE SYMBOL (if applicable) ESC/ENS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003		
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213			10. SOURCE OF FUNDING NOS.		
			PROGRAM ELEMENT NO 63756E	PROJECT NO. N/A	TASK NO N/A
			WORK UNIT NO. N/A		
11. TITLE (Include Security Classification) Mapping a Domain Model and Architecture to a Generic Design					
12. PERSONAL AUTHOR(S) A. Spencer Peterson, Jay L. Stanley, Jr.					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM TO		14. DATE OF REPORT (year, month, day) May 1994	
15. PAGE COUNT 162					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)		
FIELD	GROUP	SUB. GR.			
19. ABSTRACT (continue on reverse if necessary and identify by block number) In contrast to the number of reports on domain analysis, little work has been done in describing the utilization of domain analysis results in the development of generic designs for building applications in a domain. This report describes a process for mapping domain information in Feature-Oriented Domain Analysis (FODA) into a generic design for a domain. The design includes supporting code components that conform to the Object Connection Architecture (OCA), a model for structuring software systems. A process for the use of the design in implementing applications is included. The processes and products described herein augment the final phase of domain analysis (or engineering) <div style="text-align: right;">(please turn over)</div>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS <input checked="" type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution		
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF			22b. TELEPHONE NUMBER (include area code) (412) 268-7631		22c. OFFICE SYMBOL ESC/ENS (SEI)

ABSTRACT — continued from page one, block 19

described in the original FODA report. This report also documents the continuing work of applying FODA to the movement control domain. The design and Ada code examples for the domain used in the document are from prototype software, created in part to test the processes presented.